AIM 65

Laboratory Manual And Study Guide

LEO J. SCANLON

Rockwell
AIM 65

AIM 65

Laboratory Manual
and
Study Guide

LEO J. SCANLON
Rockwell International
Anaheim, California

JOHN WILEY & SONS NEW YORK CHICHESTER BRISBANE TORONTO

Copyright © 1981 by John Wiley & Sons, Inc.

All rights reserved.

Production or translation of any part of this work
beyond that permitted by Sections 107 or 108 of the 1976 United States Copyright Act
without the permission of the copyright owner is unlawful.
Requests for permission or further information should be addressed to
the Permissions Department, John Wiley & Sons, Inc.

ISBN 0 471 06488 2
Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

Dedicated to
my father
Leo J. Scanlon
whose intellectual curiousity has been a lifelong inspiration to me

PREFACE

Through the experiments in this book, you will receive a solid introduction to the fascinating world of microcomputers. The experience will be both instructive and "painless," because you will be working with a professional microcomputer that has been designed with education in mind—the AIM 65 Advanced Interactive Microcomputer from Rockwell International Corporation.

The "brain" of the AIM 65 *microcomputer* is a 6502 *microprocessor*, perhaps the most advanced eight-bit microprocessor on the market. The 6502 is also used in many other popular microcomputers, including the Apple II and III, the Commodore Pet, the Atari 400 and 800, and the Synertek SYM-1, as well as in a variety of industrial and consumer products.

These experiments will introduce you to principles that have importance far beyond the AIM 65 microcomputer. You will be learning about the internal architecture and instruction set of the 6502 microprocessor, many fundamental techniques of programming (including how to "debug" and design efficient programs), how external devices communicate with the microprocessor, and many other interesting and worthwhile topics.

Makeup of the Experiments . . . Most experiments in this manual have four parts. They are, in order:

Object—what you can expect to learn from the experiment.

Pre-Lab Preparation—a list of material that must be read prior to the laboratory session.

Discussion—a summary of the pertinent portions of the reading material, and additional principles that can be applied in solving the laboratory problems.

Procedure—a step-by-step approach to the problem for the experiment.

Supplementary Reference Material . . . The pre-lab reading material is drawn from three Rockwell documents that are supplied with each AIM 65 microcomputer:

AIM 65 Microcomputer User's Guide, Doc. No. 29650 N36
R6500 Programming Manual, Doc. No. 29650 N30
R6500 Hardware Manual, Doc. No. 29650 N31

Additionally, the following three Rockwell documents will prove useful to you, but they are not required for the experiments:

AIM 65 Monitor Program Listing, Doc. No. 29650 N36L

AIM 65 Summary Card, Doc. No. 29650 N51

R6500 Microprocessor Programming Reference Card, Doc. No. 29650 N50

These documents are available from: Marketing Services, Rockwell International, P.O. Box 3669, RC55, Anaheim, CA 92803

A Note About the Experiments . . . All but one of the experiments can be performed using the base model AIM 65, which has 1K bytes of RAM memory on the board. Experiment 17 requires Rockwell's optional ROM-based Assembler, and 4K bytes of RAM memory, to be present on the microcomputer board. With an Assembler-equipped AIM 65, you should perform Experiment 17 immediately after Experiment 4, and then conduct all subsequent experiments using the Assembler.

Acknowledgements . . . This book reflects the efforts of many people in addition to myself. In particular, I wish to thank the Microsystems staff of Rockwell International in Anaheim, California, for their enthusiastic support; Mrs. Lenore Seidman, for her excellent typing of the final manual; and Ms. Judy Green of John Wiley & Sons, whose many contributions helped make this book a reality. And my wife Pat deserves special thanks, for her patience and understanding during this project.

LEO J. SCANLON

CONTENTS

1 *Getting to Know the AIM 65 / 1*
2 *Addition Operations / 9*
3 *Subtraction and Logical Operations / 15*
4 *Program Sequencing / 27*
5 *Debugging Programs / 35*
6 *Multiplication Operations, with Shift & Rotate / 45*
7 *Division Operations / 55*
8 *Subroutines and the Stack / 61*
9 *Unordered Lists / 71*
10 *Sorting Unordered Data / 79*
11 *Code Conversion from Input / 85*
12 *Code Conversion for Output / 93*
13 *Input/Output / 99*
14 *A More Powerful I/O Device, the R6522 VIA / 109*
15 *Interrupts / 119*
16 *A Timing Program with Decimal Output / 129*
17 *The AIM 65 Assembler / 137*

Answers to Experiments / 147

1

GETTING TO KNOW THE AIM 65

OBJECT

To become familiar with the AIM 65, and to learn how to enter and execute a program.

PRE-LAB PREPARATION

Read Sections 1.1, 1.2, 1.9, 2.1 through 2.8 and 3.1 through 3.6 in the AIM 65 User's Guide. Read Chapter 1 and Sections 2.0 and 2.1 of Chapter 2 in the R6500 Programming Manual.

DISCUSSION

In the laboratory sessions you will be using AIM 65, the R6500 Advanced Interactive Microcomputer from Rockwell International Corporation. Before proceeding with this first experiment, take a few minutes to examine the AIM 65.

Observe that AIM 65 is a complete microcomputer, with a typewriter-style keyboard, a display, a printer and a group of electrical elements (integrated circuit "chips", resistors and capacitors). In this experiment, and those that follow, you will be using the keyboard to input data and instructions into the computer. The display and printer are output devices -- they give you visual readouts that relate to your programs.

AIM 65 has three switches. Switch S1, located on the left side of the board, is a RESET push-button. It is used to initialize the AIM 65, to put it in a known state.

Switch S2, located to the left of the display, is a two-position STEP/RUN switch. Switch S2 should be set in the STEP position for Experiment 1.

Switch S3 is a two-position TTY/KB switch that selects whether you will be using AIM 65's keyboard (KB) or an external teletypewriter (TTY). Switch S3 should be kept in the KB position for all laboratory experiments in which this manual is used.

The AIM 65 Monitor

The operating program of the AIM 65 is contained in two Read-Only Memory (ROM) chips that are installed in sockets above the keyboard's PRINT key. This program is called the Monitor. The Monitor program regulates the overall operation of the AIM 65, based on commands that you enter at the keyboard. These commands allow you to examine the contents of specific memory locations and registers (and change those contents, if desired), initiate the execution of a program, turn the printer on and off, and a variety of other system control functions. The Monitor commands are described in Section 3 of the AIM 65 User's Guide, and are listed on the AIM 65 Summary Card.

Nearly all of the Monitor commands will be used in this manual, but for Experiment 1 we will use only these eight commands:

COMMAND	DESCRIPTION
*	Alter Program Counter
I	Enter Mnemonic Instruction Entry Mode
RESET	Enter and initialize Monitor
G	Start executing user's program
K	Disassemble memory
M	Display specified memory locations
/	Alter current memory locations
R	Display register values

The R6502 Instruction Set

A microcomputer is comprised of a microprocessor (usually referred to as a CPU, or Central Processing Unit), memory and one or more input/output (I/O) devices. In the AIM 65 Microcomputer, the microprocessor is a Rockwell R6502. The R6502 is an eight-bit microprocessor, which means that it operates on instructions and data eight bits at a time. The R6502 executes programs by fetching instructions from memory, one-by-one.

Some instructions can be contained in a single eight-bit location, or byte, in memory; other instructions include an operand (a data value or a memory address), and must be contained in two or three bytes in memory. The first byte in all instructions represents an operation code, or op-code, that tells the R6502 which operation is to be performed.

The sum total of all instructions that are executeable by the R6502 is referred to as its instruction set. There are 56 instructions in the R6502 instruction set; they are summarized in Tables 1-1 and 1-2. Further, many of the 56 instructions operate with more than one addressing mode. There are 13 addressing modes in all, making the R6502 one of the most versatile and powerful microprocessors in existence. The addressing modes are discussed in detail in the R6500 Programming Manual.

When writing programs for the AIM 65, you will enter instructions in their assembly language form, using the three-letter mnemonics that make up the left-most column in Table 1-2. The AIM 65 Monitor will automatically translate each mnemonic into the proper op-code for input to the R6502. For instructions that can operate with more than one addressing mode, the Monitor will generate the proper op-code based on the format of the instruction's operand. Table 1-3 shows the general form of the operand for each of the R6502's 13 addressing modes. Each "a" in this table represents a hexadecimal character (0 through 9 and A through F), so if you were specifying an operand of hexadecimal F3 in the Immediate addressing mode, you would enter the operand in the form #F3. For example, to load F3 into the

Table 1-1. R6502 Instruction Mnemonics
(Courtesy of Rockwell International)

ADC	Add Memory to Accumulator with Carry
AND	"AND" Memory with Accumulator
ASL	Shift left One Bit (Memory or Accumulator)
BCC	Branch on Carry Clear
BCS	Branch on Carry Set
BEQ	Branch on Result Zero
BIT	Test Bits in Memory with Accumulator
BMI	Branch on Result Minus
BNE	Branch on Result not Zero
BPL	Branch on Result Plus
BRK	Force Break
BVC	Branch on Overflow Clear
BVS	Branch on Overflow Set
CLC	Clear Carry Flag
CLD	Clear Decimal Mode
CLI	Clear Interrupt Disable Bit
CLV	Clear Overflow Flag
CMP	Compare Memory and Accumulator
CPX	Compare Memory and Index X
CPY	Compare Memory and Index Y
DEC	Decrement Memory by One
DEX	Decrement Index X by One
DEY	Decrement Index Y by One
EOR	"Exclusive-or" Memory with Accumulator
INC	Increment Memory by One
INX	Increment Index X by One
INY	Increment Index Y by One
JMP	Jump to New Location
JSR	Jump to New Location Saving Return Address
LDA	Load Accumulator with Memory
LDX	Load Index X with Memory
LDY	Load Index Y with Memory
LSR	Shift One Bit Right (Memory or Accumulator)
NOP	No Operation
ORA	"OR" Memory with Accumulator
PHA	Push Accumulator on Stack
PHP	Push Processor Status on Stack
PLA	Pull Accumulator from Stack
PLP	Pull Processor Status from Stack
ROL	Rotate One Bit Left (Memory or Accumulator)
ROR	Rotate One Bit Right (Memory or Accumulator)
RTI	Return from Interrupt
RTS	Return from Subroutine
SBC	Subtract Memory from Accumulator with Borrow
SEC	Set Carry Flag
SED	Set Decimal Mode
SEI	Set Interrupt Disable Status
STA	Store Accumulator in Memory
STX	Store Index X in Memory
STY	Store Index Y in Memory
TAX	Transfer Accumulator to Index X
TAY	Transfer Accumulator to Index Y
TSX	Transfer Stack Pointer to Index X
TXA	Transfer Index X to Accumulator
TXS	Transfer Index X to Stack Register
TYA	Transfer Index Y to Accumulator

Accumulator, you would enter the instruction LDA #F3. Note that all addressing modes except the Implied mode require an operand.

Hexadecimal Number Notation

In this experiment, and most of the other experiments in this manual, we will be using the AIM 65's mnemonic instruction entry mode to enter programs into the microcomputer's memory. The mnemonic instruction entry mode requires all data values and addresses to be entered as hexadecimal characters. To eliminate the necessity of forever writing "hexadecimal F3" or "hex F3" we will henceforth use a $ prefix to denote hexadecimal values; that is, in this manual, hexadecimal F3 will be written as $F3. (The $ prefix is strictly a writing convention. The values you enter into AIM 65 must not have a $ prefix.)

Table 1-2. R6502 Instruction Set
(Courtesy of Rockwell International)

INSTRUCTIONS		IMMEDIATE			ABSOLIITE			ZERO PAGE			ACCUM			IMPLIED			(IND. X)			(INO), Y			Z PAGE. X			ABS. X			ABS. Y			RELATIVE			INDIRECT			Z PAGE. Y			PROCESSOR STATUS CODES	
MNEMONIC	OPERATION	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	7 6 5 4 3 2 1 0 N V • B D I Z C	MNEMONIC
ADC	A + M + C → A (4) (1)	69	2	2	6D	4	3	65	3	2							61	6	2	71	5	2	75	4	2	7D	4	3	79	4	3										N V • • • • Z C	ADC
AND	A ∧ M → A (1)	29	2	2	2D	4	3	25	3	2							21	6	2	31	5	2	35	4	2	3D	4	3	39	4	3										N • • • • • Z •	AND
ASL	C ← [7 0] ← 0				0E	6	3	06	5	2	0A	2	1										16	6	2	1E	7	3													N • • • • • Z C	ASL
BCC	BRANCH ON C = 0 (2)																															90	2	2							• • • • • • • •	BCC
BCS	BRANCH ON C = 1 (2)																															B0	2	2							• • • • • • • •	BCS
BEO	BRANCH ON Z = 1 (2)																															F0	2	2							• • • • • • • •	BEO
BIT	A ∧ M				2C	4	3	24	3	2																															M_7 M_6 • • • • Z •	BIT
BMI	BRANCH ON N = 1 (2)																															30	2	2							• • • • • • • •	BMI
BNE	BRANCH ON Z = 0 (2)																															D0	2	2							• • • • • • • •	BNE
BPL	BRANCH ON N = 0 (2)																															10	2	2							• • • • • • • •	BPL
BRK	BREAK													00	7	1																									• • • 1 • 1 • •	BRK
BVC	BRANCH ON V = 0 (2)																															50	2	2							• • • • • • • •	BVC
BVS	BRANCH ON V = 1 (2)																															70	2	2							• • • • • • • •	BVS
CLC	0 → C													18	2	1																									• • • • • • • 0	CLC
CLD	0 → D													D8	2	1																									• • • • 0 • • •	CLD
CLI	0 → I													58	2	1																									• • • • • 0 • •	CLI
CLV	0 → V													B8	2	1																									• 0 • • • • • •	CLV
CMP	A − M	C9	2	2	CD	4	3	C5	3	2							C1	6	2	D1	5	2	D5	4	2	DD	4	3	D9	4	3										N • • • • • Z C	CMP
CPX	X − M	E0	2	2	EC	4	3	E4	3	2																															N • • • • • Z C	CPX
CPY	Y − M	C0	2	2	CC	4	3	C4	3	2																															N • • • • • Z C	CPY
DEC	M − 1 → M				CE	6	3	C6	5	2													D6	6	2	DE	7	3													N • • • • • Z •	DEC
DEX	X − 1 → X													CA	2	1																									N • • • • • Z •	DEX
DEY	Y − 1 → Y													88	2	1																									N • • • • • Z •	DEY
EOR	A ⊻ M → A (1)	49	2	2	4D	4	3	45	3	2							41	6	2	51	5	2	55	4	2	5D	4	3	59	4	3										N • • • • • Z •	EOR
INC	M + 1 → M				EE	6	3	E6	5	2													F6	6	2	FE	7	3													N • • • • • Z •	INC
INX	X + 1 → X													E8	2	1																									N • • • • • Z •	INX
INY	Y + 1 → Y													C8	2	1																									N • • • • • Z •	INY
JMP	JUMP TO NEW LOC				4C	3	3																												6C	5	3				• • • • • • • •	JMP
JSR	JUMP SUB				20	6	3																																		• • • • • • • •	JSR
LDA	M → A (1)	A9	2	2	AD	4	3	A5	3	2							A1	6	2	B1	5	2	B5	4	2	BD	4	3	B9	4	3										N • • • • • Z •	LDA
LDX	M → X (1)	A2	2	2	AE	4	3	A6	3	2																			BE	4	3							B6	4	2	N • • • • • Z •	LDX
LDY	M → Y (1)	A0	2	2	AC	4	3	A4	3	2													B4	4	2	BC	4	3													N • • • • • Z •	LDY
LSR	0 → [7 0] → C				4E	6	3	46	5	2	4A	2	1										56	6	2	5E	7	3													0 • • • • • Z C	LSR
NOP	NO OPERATION													EA	2	1																									• • • • • • • •	NOP
ORA	A ∨ M → A	09	2	2	0D	4	3	05	3	2							01	6	2	11	5	2	15	4	2	1D	4	3	19	4	3										N • • • • • Z •	ORA
PHA	A → Ms S − 1 → S													4B	3	1																									• • • • • • • •	PHA
PHP	P → Ms S − 1 → S													08	3	1																									• • • • • • • •	PHP
PLA	S + 1 → S Ms → A													68	4	1																									N • • • • • Z •	PLA
PLP	S + 1 → S Ms → P													2B	4	1																									(RESTORED)	PLP
ROL	[7 0] ← [C]				2E	6	3	26	5	2	2A	2	1										36	6	2	3E	7	3													N • • • • • Z C	ROL
ROR	[C] → [7 0]				6E	6	3	66	5	2	6A	2	1										76	6	2	7E	7	3													N • • • • • Z C	ROR
RTI	RTRN INT													40	6	1																									(RESTORED)	RTI
RTS	RTRN SUB													60	6	1																									• • • • • • • •	RTS
SBC	A − M − $\overline{C}$ → A (1)	E9	2	2	ED	4	3	E5	3	2							E1	6	2	F1	5	2	F5	4	2	FD	4	3	F9	4	3										N V • • • • Z (3)	SBC
SEC	1 → C													38	2	1																									• • • • • • • 1	SEC
SED	1 → D													F8	2	1																									• • • • 1 • • •	SED
SEI	1 → I													7B	2	1																									• • • • • 1 • •	SEI
STA	A → M				8D	4	3	B5	3	2							81	6	2	91	6	2	95	4	2	9D	5	3	99	5	3										• • • • • • • •	STA
STX	X → M				8E	4	3	86	3	2																												96	4	2	• • • • • • • •	STX
STY	Y → M				BC	4	3	84	3	2													94	4	2																• • • • • • • •	STY
TAX	A → X													AA	2	1																									N • • • • • Z •	TAX
TAY	A → Y													A8	2	1																									N • • • • • Z •	TAY
TSX	S → X													BA	2	1																									N • • • • • Z •	TSX
TXA	X → A													BA	2	1																									N • • • • • Z •	TXA
TXS	X → S													9A	2	1																									• • • • • • • •	TXS
TYA	Y → A													98	2	1																									N • • • • • Z •	TYA

(1) ADD 1 to "N" IF PAGE BOUNDARY IS CROSSED
(2) ADD 1 TO "N" IF BRANCH OCCURS TO SAME PAGE
ADD 2 TO "N" IF BRANCH OCCURS TO DIFFERENT PAGE
(3) CARRY NOT = BORROW
(4) IF IN DECIMAL MODE, Z FLAG IS INVALID
ACCUMULATOR MUST BE CHECKED FOR ZERO RESULT

X	INDEX X	+	ADD	M_7	MEMORY BIT 7
Y	INDEX Y	−	SUBTRACT	M_6	MEMORY BIT 6
A	ACCUMULATOR	∧	AND	n	NO. CYCLES
M	MEMORY PER EFFECTIVE ADDRESS	∨	OR	#	NO. BYTES
Ms	MEMORY PER STACK POINTER	⊻	EXCLUSIVE OR		

ASSIGNMENT

Develop a program that will load a value of $AC into the Accumulator and then store the contents of the Accumulator in location $40. The program should start at location $0200. To do this, you should use these two instructions:

INSTRUCTION	DESCRIPTION
LDA	Load Accumulator with Memory
STA	Store Accumulator in Memory

Table 1-3. The 6502 Addressing Modes

Mode	Operand Format
Immediate	#aa
Absolute	aaaa
Zero Page	aa
Implied	
Indirect Absolute	(aaaa)
Absolute Indexed, X	aaaa,X or aaaaX
Absolute Indexed, Y	aaaa,Y or aaaaY
Zero Page Indexed, X	aa,X or aaX
Zero Page Indexed, Y	aa,Y or aaY
Indexed Indirect	(aa,X) or (aaX)
Indirect Indexed	(aa),Y or (aa)Y
Relative	aa or aaaa
Accumulator	A

PROCEDURE

1. Turn on power to your AIM 65. What are the contents of the Program Counter and the Accumulator for this "initial" condition?

   PC= A=

2. Which Monitor command did you use to determine the register values in Step 1?

3. What are the current contents of memory location $40?

   Location $40 contains

4. Which Monitor command did you use to determine the contents of location $40 in Step 3?

5. Write a two-instruction program that will load the value $AC into the Accumulator and then store the contents of the Accumulator in location $40.

6. Which addressing mode was used in your load instruction?

7. Which addressing mode was used in your store instruction?

8. Change the contents of the Program Counter so that it points to location $0200, the starting address of your program. Which Monitor command did you use to load $0200 into the Program Counter?

9. Place the AIM 65 in the mnemonic instruction entry mode. Which Monitor command did you use to accomplish this?

10. What is the display showing at this point?

    What does this display indicate?

11. Enter your program from Step 5 into the computer, and press ESC when you are done.

12. What is the contents of the Program Counter now?

    Explain why the Program Counter has this value.

13. Change the Program Counter so that it addresses the starting location of your program, then execute the program. Which Monitor command did you use to execute the program?

14. What do you expect to find in the Program Counter, the Accumulator and memory location $40 at this point?

    PC= A= $40=

    Verify your answers, using the proper Monitor commands.

    PC= A= $40=

15. If you programmed the problem in the most efficient way, your two instructions should occupy four bytes in memory, locations $0200 through $0203. List the contents of each of these locations below, and provide a short description of what each location contains.

LOCATION	HEX.CONTENTS	DESCRIPTION OF CONTENTS
$0200		
$0201		
$0202		
$0203		

2

ADDITION OPERATIONS

OBJECT

To learn how to perform addition operations on signed and unsigned integer numbers.

PRE-LAB PREPARATION

Read Sections 6.2 and 6.8 of the AIM 65 User's Guide, and Section 2.2.1 and Chapter 3 of the R6500 Programming Manual.

DISCUSSION

The R6502 instruction set contains only one addition instruction, Add Memory to Accumulator with Carry (ADC). This instruction adds the contents of a specified memory location (or the second byte of the instruction, if Immediate addressing is used) and the contents of the Status Register's Carry bit to the Accumulator. The ADC instruction can be used to perform addition operations on three types of integer numbers: unsigned and signed binary numbers, and Binary-Coded-Decimal (BCD) numbers.

Status Flags for Addition Operations

The ADC instruction affects four flags in the Processor Status Register:

- The Carry (C) flag is set if the sum of a binary addition exceeds decimal 255 ($FF) or if the sum of a BCD addition exceeds decimal 99 ($99); otherwise it is reset.
- The Zero (Z) flag is set if the sum is zero; otherwise it is reset.
- The Negative (N) flag is set if Bit 7 of the sum is a logic one; otherwise it is reset. The state of the N flag is meaningful only if signed numbers are being added, in which case it indicates whether the result is negative (N=1) or positive (N=0).
- The Overflow (V) flag is set if two like-signed numbers (both positive or both negative) are added and the sum exceeds +127 ($7F) or -128 ($80), which causes Bit 7 of the Accumulator to be changed; otherwise V is reset.

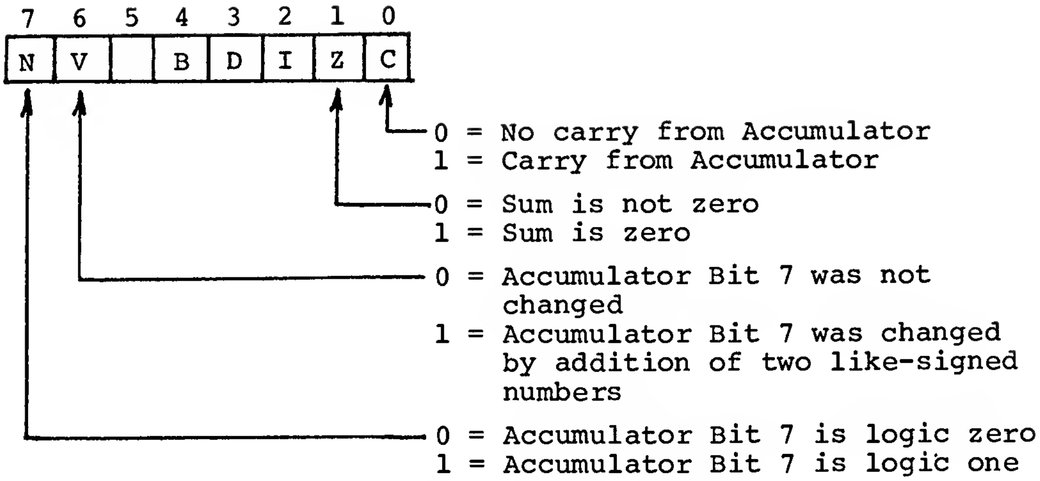

7 6 5 4 3 2 1 0
N V B D I Z C
0 = No carry from Accumulator
1 = Carry from Accumulator
0 = Sum is not zero
1 = Sum is zero
0 = Accumulator Bit 7 was not changed
1 = Accumulator Bit 7 was changed by addition of two like-signed numbers
0 = Accumulator Bit 7 is logic zero
1 = Accumulator Bit 7 is logic one

Figure 2-1. Status Flags for Addition

R6502 Instructions Used for Addition

You will be using the following new R6502 instructions in this experiment:

INSTRUCTION	DESCRIPTION
ADC	Add Memory to Accumulator with Carry
CLC	Clear Carry Flag
SED	Set Decimal Mode
CLD	Clear Decimal Mode

The ADC instruction adds a specified memory location and the contents of the Status Register's Carry flag to the Accumulator. The automatic addition of Carry is handy in multiple-byte additions, but it also forces the programmer to clear Carry (using CLC) before the first ADC instruction is executed.

The decimal mode instructions, SED and CLD, put the R6502's adder into the decimal mode and restore it to the binary mode, respectively.

The CLC, SED and CLD instructions require no operand -- their functions are fully defined by the mnemonic. For example, the Clear Carry Flag (CLC) instruction performs a specific operation (clearing) on the Status Register's Carry flag; the instruction needs no additional variable to define its operation. These types of instructions use the Implied addressing mode. Nearly half of the R6502 instructions operate with Implied addressing.

PROCEDURE

1. Write a program to add the unsigned integer contents of memory locations $0200 and $0201, storing the sum in memory location $0202.

   Draw the flowchart below.

   Write the program code below.

Run your program for the three cases listed below, and complete the table.

$0200		$0201		$0202		Carry? (Yes/No)
Dec.	Hex.	Dec.	Hex.	Dec.	Hex.	
143	8F	131	83	274		
46	2E	65	41	111		
59	3B	197	C5	256		

2. Write a program to add the signed contents of memory locations $0200 and $0201, storing the sum in location $0202.

   Will the flowchart and/or program code differ from that of Step 1? If so, describe the differences below.

Run your program for the cases listed below, and complete the table.

$0200		$0201		$0202		Overflow? (Yes/No)
Dec.	Hex.	Dec.	Hex.	Dec.	Hex.	
106	6A	89	59	195		
127	7F	3	03	130		
32	20	-3	FD	29		
-90	A6	-62	C2	-152		

3. Write a program to add two 16-bit (two-byte) signed numbers, one stored in locations \$0200 and \$0201, the other stored in locations \$0202 and \$0203. The sum should be returned in locations \$0204 and \$0205. Numbers should be stored with the low-order byte in the lower-addressed locations; e.g., \$0200 holds the least-significant byte of the first addend.

\$0200	LSB_A
\$0201	MSB_A
\$0202	LSB_B
\$0203	MSB_B
\$0204	LSB_{Sum}
\$0205	MSB_{Sum}

Flowchart the problem in the space below.

Write the program code below.

Run your program for the two cases listed below, and complete the table.

Addend #1 $0200,$0201	Addend #2 $0202,$0203	Sum $0204,$0205	Overflow? (Yes/No)
$2563	$41AB		
$83C7	$F1B4		

4. Using AIM 65, how would you determine whether a carry occurred between bytes during a multi-byte addition, such as the one performed in Step 3?

3

SUBTRACTION AND LOGICAL OPERATIONS

OBJECT

To become familiar with subtraction operations on signed and unsigned numbers, and the use of the AND, OR and XOR logical operations.

PRE-LAB PREPARATION

Read Sections 2.2.2, 2.2.3 and 2.2.4 of the R6500 Programming Manual, and Section 6.3 of the AIM 65 User's Guide.

DISCUSSION

Subtraction Operations

The R6502 instruction set contains only one subtraction instruction, Subtract Memory from Accumulator with Carry (SBC). This instruction subtracts the contents of the specified memory location (or the second byte of the instruction, if Immediate addressing is used) and the contents of the Status Register's Carry bit (in inverted form) from the Accumulator.

Why is Carry used in its inverted form? Because for subtraction, Carry represents a borrow; a final Carry = 1 indicates absence of borrow, Carry = 0 indicates that the subtraction has produced a borrow. Whereas Carry must be initialized to 0 for addition operations, it must be initialized to 1 for subtraction operations.

Status Flags for Subtraction Operations

The SBC instruction affects four flags in the Processor Status Register:

- The Carry (C) flag is set if the result is positive or zero, and is reset if the result is negative (indicating a borrow).
- The Zero (Z) flag is set if the result is zero; otherwise it is reset. Note that if Carry and Zero are both set, the result is zero. If Carry is set and Zero is reset, the result is positive.

- The Negative (N) flag is set if Bit 7 of the result is a logic one; otherwise it is reset. The state of the N flag is meaningful only if signed numbers are being subtracted, in which case it indicates whether the result is negative (N=1) or positive (N=0).
- The Overflow (V) flag is set if two unlike-signed numbers (one number positive, the other number negative) are subtracted and the result exceeds $+127_{10}$ or -128_{10}, which causes Bit 7 of the Accumulator to be changed.

R6502 Instructions Used for Subtraction

You will be using the following new R6502 instructions in the subtraction portion of this experiment

INSTRUCTION	DESCRIPTION
SBC	Subtract Memory from Accumulator with Borrow
SEC	Set Carry Flag

The SBC instruction subtracts a specified memory location and the contents of the Status Register's Carry flag from the Accumulator. The automatic subtraction of Carry is handy in multiple-byte subtractions, but it also forces the programmer to set Carry (using SEC) before the first SBC instruction is executed.

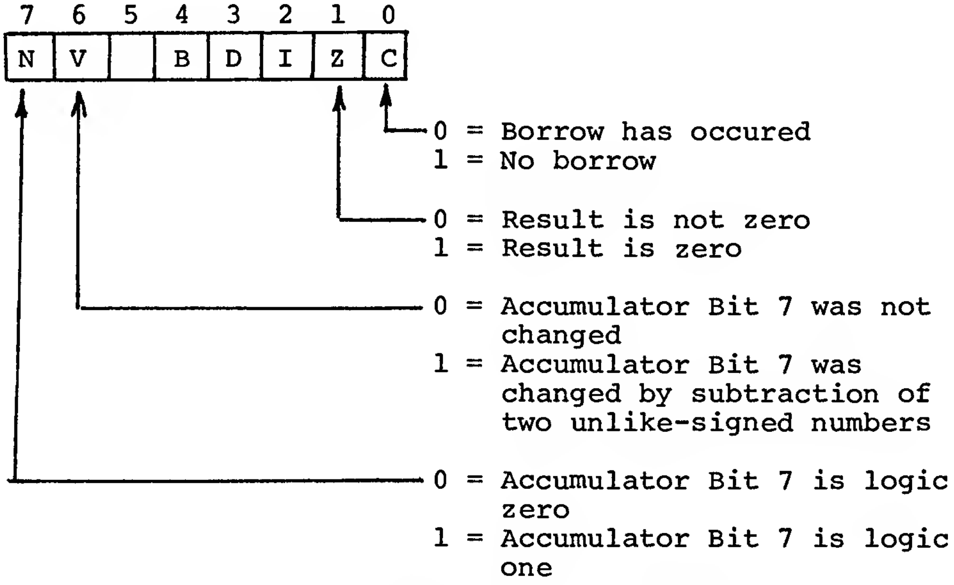

7 6 5 4 3 2 1 0
N V B D I Z C
0 = Borrow has occured
1 = No borrow
0 = Result is not zero
1 = Result is zero
0 = Accumulator Bit 7 was not changed
1 = Accumulator Bit 7 was changed by subtraction of two unlike-signed numbers
0 = Accumulator Bit 7 is logic zero
1 = Accumulator Bit 7 is logic one

Figure 3-1. Status Flags for Subtraction

Logical Operations

The R6502 supports three types of logical operations: AND, OR and Exclusive-OR. All three types operate on the Accumulator, using the contents of a memory location or, with Immediate addressing, the contents of the second byte of the instruction.

An AND operation produces a 1 in each Accumulator bit position in which both memory and the Accumulator contain a 1; otherwise that bit position is reset to 0.

Table 3-1. Logical AND Operation

Memory Bit	Accumulator Bit	Result in Accumulator
0	0	0
0	1	0
1	0	0
1	1	1

An OR operation produces a 1 in each Accumulator bit position in which either memory or the Accumulator (or both) contain a 1.

Table 3-2. Logical OR Operation

Memory Bit	Accumulator Bit	Result in Accumulator
0	0	0
0	1	1
1	0	1
1	1	1

An Exclusive-OR operation produces a 1 in each Accumulator bit position in which either memory or the Accumulator (but not both) contain a 1.

Table 3-3. Logical Exclusive-OR Operation

Memory Bit	Accumulator Bit	Result in Accumulator
0	0	0
0	1	1
1	0	1
1	1	0

R6502 Instructions Used for Logical Operations

The R6502 has three logical instructions:

INSTRUCTION	DESCRIPTION
AND	AND Memory with Accumulator
ORA	OR Memory with Accumulator
EOR	Exclusive-OR Memory with Accumulator

PROCEDURE

1. Write a program to subtract the 24-bit signed binary number in memory locations $0200, $0201 and $0202 (most-significant byte in $0202) from the 24-bit signed binary number in memory locations $0203, $0204 and $0205. Store the result in locations $0203 through $0205, replacing the minuend. Write the program code below. The structure of this program will be similar to that of the double precision addition program you developed in Steps 3 and 4 of Experiment 2.

2. Run your program to perform this subtraction:

   $267BFE - $74A017

   List the contents of the operand locations in this table.

Subtrahend			Minuend		
$0200	$0201	$0202	$0203	$0204	$0205

List the result in this table:

Results			
$0203	$0204	$0205	Carry =

Is the result in locations $0203 through $0205 valid? What did you check to determine whether or not the result is valid?

3. The preceding problem involved three successive two-byte subtractions. How can you determine which of these subtractions generated a borrow, without performing the subtraction by hand? (Hint: Recall that Carry = 0 indicates a borrow.)

Using your proposed method, determine the total number of borrows that occurred, and record the answer below.

Total number of borrows =

4. Consider an industrial control system that has eight devices, listed in Table 3-4. A group of eight memory bits reflect the status of the eight devices at any given point in time.

Table 3-4. Logic Information for System of Eight Devices

BIT POSITION	DEVICE	LOGIC STATE INFORMATION
0	Pressure sensor	1 = Pressure above setpoint 0 = Pressure at or below setpoint
1	Temperature sensor	1 = Temperature above setpoint 0 = Temperature at or below setpoint
2	Velocity sensor	1 = Velocity above setpoint 0 = Velocity at or below setpoint
3	Flow rate sensor	1 = Flow rate above setpoint 0 = Flow rate at or below setpoint
4	Concentration sensor	1 = Concentration above setpoint 0 = Concentration at or below setpoint
5	Valve A	1 = Valve A open 0 = Valve A closed
6	Valve B	1 = Valve B open 0 = Valve B closed
7	Power	1 = Power on 0 = Power off

For example, a status byte with the binary configuration 11100010 signifies that pressure is at or below the setpoint, temperature is above the setpoint, velocity is at or below the setpoint, and so on. Write a program to monitor the eight devices in the system, based on the relationship of two status bytes. The Current Status Byte (CSB) is stored at location $0300. The Prior Status Byte (PSB) is stored at location $0301.

Your program should do the following:

a. Check to see which devices have changed state (Exclusive-OR operation)

b. Store the result of Step a. in location $0302.

4. (Cont'd.)

   c. Determine which devices have changed from logic 1 to logic 0 (AND operation).

   d. Store the result of Step c. in location $0303.

   e. Determine which devices have changed from logic 0 to logic 1 (complement, then an AND operation). Hint: The instruction EOR # $FF will complement the Accumulator.

   f. Store the result of Step e. in location $0304.

LOCATION	CONTENTS
$0300	Current Status Byte ($0300)
$0301	Prior Status Byte ($0301)
$0302	Bits that changed state
$0303	Bits that changed from 1 to 0
$0304	Bits that changed from 0 to 1

Flowchart the program on the next page.

4. (Cont'd.)

Write the source code below.

5. Referring to the problem in Step 4, assume that the Current Status Byte has been found to be:

$CSB = 11101010_2$

and the Prior Status Byte is

$PSB = 11101001_2$

List the physical state of each device (above setpoint, etc.) in Table 3-5 for both the CSB and PSB.

Table 3-5. Summary of CSB and PSB

Device	PSB	CSB
Pressure Sensor		
Temperature Sensor		
Velocity Sensor		
Flow Rate Sensor		
Concentration Sensor		
Valve A		
Valve B		
Power		

Run your program for the above CSB and PSB conditions and list the results below.

Devices that changed: ______________________________

Devices changed from 1 to 0: ______________________________

Devices changed from 0 to 1: ______________________________

4

PROGRAM SEQUENCING

OBJECT

To become familiar with loop and branch operations.

PRE-LAB PREPARATION

Read Chapter 4 of the R6500 Programming Manual and Sections 6.4 and 6.5 of the AIM 65 User's Guide.

DISCUSSION

Computer programs are sequential by nature, in that they carry out some specified operation by performing a series of instructions one after the other. It is important to realize, however, that the order in which the program appears on a coding sheet -- and within the computer's memory -- can be quite different from the sequence in which the instructions are executed by the microprocessor.

The microprocessor, hereafter referred to as the CPU (Central Processing Unit), takes program instructions from memory based on a special "instruction address" contained in the Program Counter (PC). At the beginning of a program, the Program Counter will contain the memory address of the program's first instruction. The CPU will use this address to fetch that instruction, and update the Program Counter to address the next instruction in memory.

The R6502 has two types of instructions whose sole purpose is to control the program sequence by altering (or not altering) the Program Counter.

- Branch instructions, which alter the Program Counter by an offset relative to the Branch instruction. Branch instructions are conditional on the status of selected bits in the Status Register -- that is, the branch is taken only if the condition is satisfied.
- Jump instructions, which load a new address into the Program Counter. Jump instructions are unconditional -- the jump is taken (i.e., the Program Counter is altered) whenever the Jump instruction is executed.

R6502 Instructions for Branches and Jumps

The following Branch and Jump instructions apply to this experiment:

INSTRUCTION	DESCRIPTION
BCC	Branch on Carry Clear (C=0)
BCS	Branch on Carry Set (C=1)
BEQ	Branch on Result Zero (Z=1)
BMI	Branch on Result Minus (N=1)
BNE	Branch on Result Not Zero (Z=0)
BPL	Branch on Result Plus (N=0)
BVC	Branch on Overflow Clear (V=0)
BVS	Branch on Overflow Set (V=1)
JMP	Jump to New Location

One additional Jump instruction, JSR (Jump to Subroutine), will be covered in Experiment 8.

The R6502 Compare Instructions

The R6502 provides instructions that permit you to compare the contents of the Accumulator, X Register or Y Register with a value in memory without altering the contents of the register.

These instructions are:

INSTRUCTION	DESCRIPTION
CMP	Compare Memory and Accumulator
CPX	Compare Memory and X Register
CPY	Compare Memory and Y Register

Essentially, these instructions subtract the contents of the memory byte from the specified register, and record the result of the operation in three flags in the Status Register:

- o The Carry (C) flag is set if the register is greater than or equal to memory, and reset if the register is less than memory.
- o The Zero (Z) flag is set if the contents of the register and memory are identical, otherwise it is reset.
- o The Negative (N) flag is meaningful only when signed numbers are being compared.

N is set if the register is less than memory, and is reset if the register is greater than or equal to memory. Table 4-1 summarizes the compare flags.

Table 4-1. Compare Instruction Results

Condition	N*	Z	C
A, X or Y less than memory	1	0	0
A, X or Y equal to memory	0	1	1
A, X or Y greater than memory	0	0	1

*N is valid only for signed compares.

Bit Test Instruction

The BIT instruction performs a logical AND between the addressed memory byte and the Accumulator, and has the following effects on the Status Register:

- The Zero (Z) flag is set if memory and the Accumulator have no "one" bits in common, and reset otherwise.
- The Overflow (V) flag receives the value of Bit 6 of the addressed memory byte.
- The Negative (N) flag receives the value of Bit 7 of the addressed memory byte.

Instructions to Increment/Decrement Registers and Memory

The compare and branch instructions are often used in program loops, in which an operation is repeated a specified number of times. In such cases, the loop count is maintained in either a memory location or a register (X or Y). The R6502 instructions to increment or decrement memory or register contents are:

INSTRUCTION	DESCRIPTION
DEC	Decrement Memory by One
DEX	Decrement X Register by One
DEY	Decrement Y Register by One
INC	Increment Memory by One
INX	Increment X Register by One
INY	Increment Y Register by One

The Jump (JMP) instruction can operate with two different addressing modes, Absolute and Indirect Absolute, so the operand must be a four-digit hexadecimal address in either this Absolute form:

```
JMP   aaaa
```

or this Indirect Absolute form:

```
JMP   (aaaa)
```

The Branch instructions (Bnn, in general form) operate only with the Relative addressing mode; the operand must be an 8-bit signed offset, relative to the second byte of the Branch instruction. However, the AIM 65 Monitor will, optionally, accept a four-digit absolute address as the Branch instruction operand, and use this address to calculate the relative offset for you! Therefore, you can specify an operand in either this Relative form:

```
Bnn     aa
```

or this Absolute form:

```
Bnn     aaaa
```

Since AIM 65 prints out the instruction addresses as you type a program into the computer, backward Jump or Branch addresses are readily available from the print-out. How do you know which address to specify if the "target" instruction is past the Jump or Branch instruction that you are typing in? Do not worry about it! Just take a rough guess at the target instruction's address, and type that address in. After you have entered the entire program, and have seen the correct target address(es) printed out, you can come back and replace the "guess" Jump or Branch instruction with an instruction having the correct operand.

Program Terminators

In the preceding experiments, you have executed programs in only one way: by entering a G command (i.e., pressing the G key), followed by a two-digit number that specifies how many instructions are to be executed. For programs with no jumps or branches, the two-digit instruction count is simply the number of instructions in the program. For example, G/07 was entered to run a program that was seven instructions long. However, programs having branch instructions will vary depending on whether or not a branch is taken. That is, perhaps seven instructions will be executed under one set of input parameters and nine instructions will be executed under a different set of input parameters.

From this point on in the manual you should add a special program terminator instruction to the end of your program, so you need not count the instructions to be executed unless you choose to do so. A program terminator will allow you to execute programs with the command

```
G/.
```

which tells the R6502, "Run the program until you encounter a program terminator."

If you are running a program with AIM 65 in the Step mode, a good terminator to use is

```
JMP  E1A1
```

which transfers control to the AIM 65 Monitor. With AIM 65 in the Run mode, you can use either JMP E1A1, or this instruction

INSTRUCTION	DESCRIPTION
BRK	Force Break

which will also transfer control to the Monitor (but only in Run mode).

PROCEDURE

1. Write a program to arrange two single-byte, unsigned numbers in order of increasing value. The numbers are contained in locations $20 and $21. Upon completion of the program, location $22 should reflect the outcome as follows:

   o ($22) = 0 if ($20) is less than ($21)

   o ($22) = 1 if ($20) is equal to ($21)

   o ($22) = 2 if ($20) is greater than ($21)

   Use the X Register for temporary storage if an exchange is required. Figure 4-1 is a flowchart for the program. Write the program code below. The last instruction should be JMP E1A1.

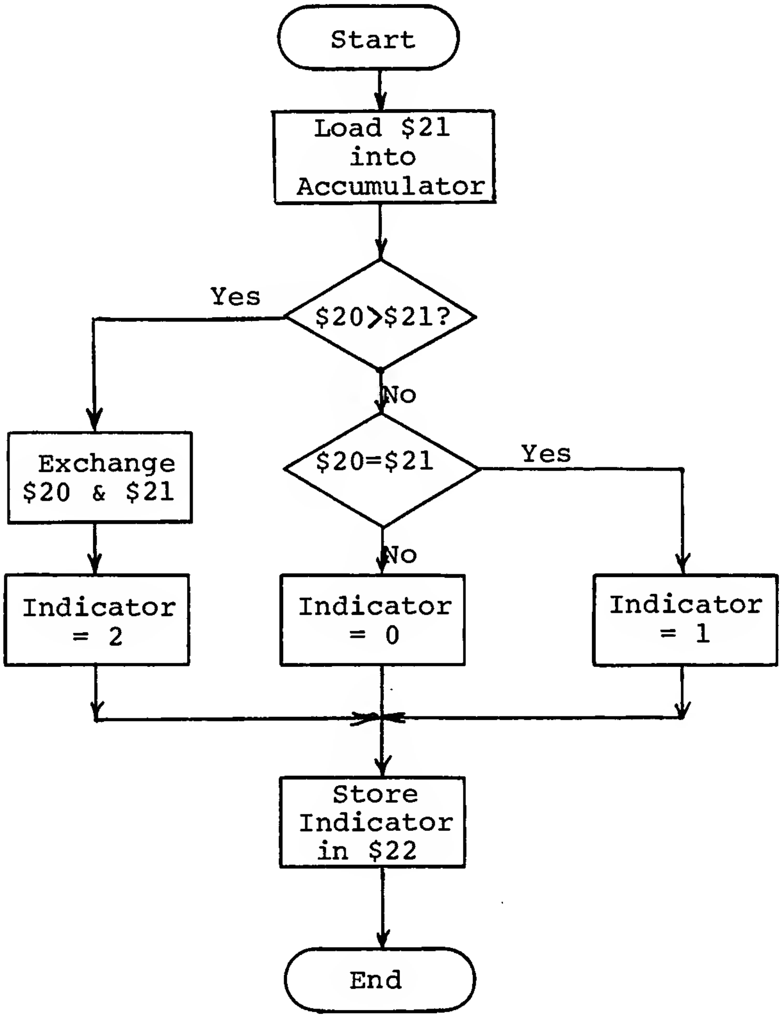

Start
Load $21 into Accumulator
$20>$21?
Yes
No
Exchange $20 & $21
$20=$21
Yes
No
Indicator = 2
Indicator = 0
Indicator = 1
Store Indicator in $22
End

Figure 4-1. Flowchart to Arrange Two Byte Values in Increasing Order

2. Run your program for the values given below, and complete the $22 column for each case. You should also use the AIM 65 Monitor's M command to check the final contents of locations $20 and $21.

$20		$21		$22
Dec.	Hex.	Dec.	Hex.	Hex.
150	96	14	0E	
150	96	150	96	
0	0	40	28	
0	0	0	0	

3. Find the largest unsigned number in memory locations $21 through $2A and place it in memory location $20. Flowchart the problem in the space below.

Write the program code below. Use indexed addressing to process the data table.

4. Run your program for the data table values given below, and record the result in the space provided.

Location	Contents
$21	$A2
$22	$00
$23	$40
$24	$40
$25	$CE
$26	$5A
$27	$FA
$28	$55
$29	$06
$2A	$4F

Result in $20 =

5

DEBUGGING PROGRAMS

OBJECT

To learn how the AIM 65 Monitor's trace and breakpoint features can be used to identify programming errors.

PRE-LAB PREPARATION

Read Sections 3.6 and 3.7 in the AIM 65 User's Guide.

DISCUSSION

In the preceding lab experiments, each of the programming assignments involved drawing a flowchart of the program, writing the instruction sequence in the lab manual, then entering these instructions (the source code) into AIM 65 memory with the I command and executing them with the G Command.

Unless all of your programs performed correctly the first time through, you had to do some "debugging", to get the errors out of the program. Until now, the easiest way to find errors was to run a portion of the program -- perhaps the first few instructions -- and display the contents of registers and meaningful memory locations at that point in the program. If the displayed values were not what you expected, you searched the preceding instructions to find out "what went wrong."

The procedure just described will do the job for very small or very simple programs, but becomes time-consuming and inefficient for debugging more complex programs. The AIM 65 Monitor includes two command types that are specifically designed as debugging aids: they are the trace commands and the breakpoint commands.

IMPORTANT!!

Both command types can only be used when the AIM 65's RUN/STEP switch is in the STEP position.

Trace Commands

As you know, the Monitor's G command causes the R6502 CPU to execute the specified number of instructions, then return to the Monitor command mode. The G command generates no printed information in the course of the execution sequence. This lack of print-out is inconsequential if your program is error-free, and working as you intended it to work, but can prove a real hindrance if you are trying to track down errors in the program. The trace commands can be used to provide the kind of printed information you need for debugging a program. There are three trace commands:

COMMAND	DESCRIPTION
Z	Toggle Instruction Trace Mode On/Off
V	Toggle Register Trace Mode On/Off
H	Trace Program Counter History

The Z command causes AIM 65 to disassemble and print each instruction in a program before it is executed. Pressing the Z key turns this feature off if it was on, and on if it was off, and displays the resulting on/off status.

The V command causes AIM 65 to print the register contents after each instruction is executed. Register contents are printed in the same format produced by the R command, but without a labeled header (normally, an R command is given before the G command, to provide the header for the V command's listings). Pressing the V key turns the register trace feature off if it was on, and on if it was off, and displays the resulting on/off status.

There are many situations in which you don't need to know all of the information provided by the Z or V command, but simply want to know the execution path the program has taken. For these situations, you can use the H command, which prints out the Program Counter addresses of the last four instructions executed, and the address of the next instruction to be executed.

Breakpoint Commands

The AIM 65 Monitor's breakpoint feature is perhaps the most powerful debugging option at your disposal. With this feature, you can assign up to four different instruction addresses in your program as "breakpoint" addresses. Here is how the breakpoint feature operates: With the AIM 65 in the Step mode, the R6502 will halt each time it is about to execute an instruction whose address has been specified as a breakpoint. At this time, the disassembled form of the instruction will be displayed and printed, and the R6502 will return to the Monitor.

With the R6502 halted, the contents of any memory location can be examined with the M command, and register contents can be examined with the R command. In this way, the breakpoint feature provides you with the opportunity to investigate meaningful parameters at selected points in a program. Further, since the R6502 will halt on a breakpoint only if it encounters that breakpoint, this feature can also be used to check whether or not a certain program instruction is ever being executed!

There are four breakpoint commands:

COMMAND	DESCRIPTION
#	Clear All Breakpoints
4	Toggle Breakpoint Enable On/Off
B	Set/Clear Breakpoint Address
?	Display Breakpoint Addresses

The AIM 65 breakpoints are in an undefined state when power is turned on, and should be cleared (reset, to address $0000) by pressing the # key.

From this initial state, you can assign a breakpoint by pressing the B key. When the AIM 65 displays the prompt BRK/, enter the number of the breakpoint to be assigned (0, 1, 2 or 3), followed by the four-digit hexadecimal address of the appropriate instruction. Repeat this procedure for each breakpoint that you want to assign.

When all desired breakpoint addresses have been assigned, turn the breakpoint feature on, with the 4 key, then initialize the Program Counter to the program's starting address and enter G/. to execute the program. The program will run until it encounters a breakpoint address, at which time it will halt and display the instruction at that address -- the next instruction to be executed. With the R6502 halted, you can examine memory and register contents, as described previously. If all values are as expected, enter G/. to resume execution; if not, modify your program as required (remember, the AIM 65 is in the Monitor command mode after encountering a breakpoint).

Once you get to the point where you feel that all errors have been corrected, you can disable (turn off) the breakpoint feature by pressing the 4 key, and execute the program from the beginning. Disabling the breakpoint feature does not affect the assigned breakpoint addresses; the breakpoints will remain intact until you re-assign them, or turn power off.

Where should breakpoints be placed in a program? A few of the likely places include these:

1. Following a loop -- to investigate counter values and results in memory and registers.
2. At branch destination instructions -- to see if and when the branch occurs.
3. After a complex calculation -- to check the result before continuing.
4. After inputting data from a peripheral device -- to see the data value that has been received.

The exercise in this experiment will demonstrate how traces and breakpoints are used to find some common types of errors in a program. It will also give you a good groundwork for using these features to debug programs in the future.

PROCEDURE

1. In this experiment, we will develop a program that counts the number of zeroes in memory location \$21 through \$2A, and places that count in location \$20. Figures 5-1 and 5-2 show a flowchart and a symbolic program that should do the job.

2. Since the AIM 65 will not accept symbolic labels without the use of an assembler, the program in Figure 5-2 must be converted into hexadecimal form. If we store the program into memory starting at location \$0200, the hexadecimal, mnemonic entry form of the program is as shown in Figure 5-3.

3. Let us assume for the moment that the program in Figure 5-3 is error free, and see what results a trial run produces. Enter the program into the AIM 65, then initialize memory locations \$20 through \$2A with \$00.

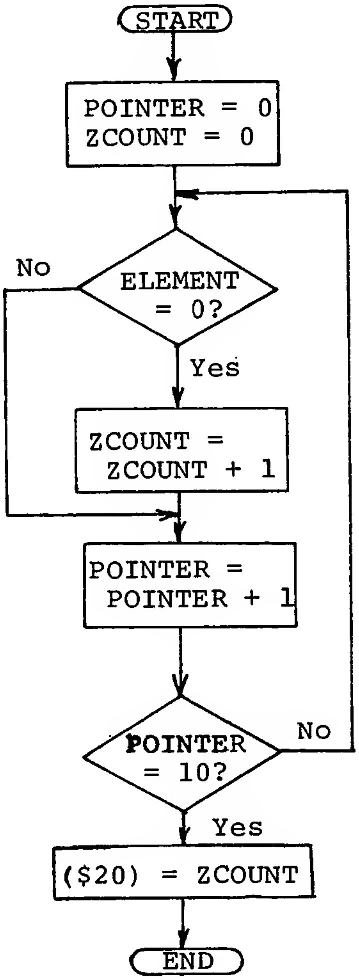

START
POINTER = 0
ZCOUNT = 0
No
ELEMENT = 0?
Yes
ZCOUNT = ZCOUNT + 1
POINTER = POINTER + 1
POINTER = 10?
No
Yes
($20) = ZCOUNT
END

Figure 5-1. Flowchart for Zeroes-Counting Program

```
0000    *=0200
0200 A2 LDX #00
0202 A0 LDY #00
0204 B5 LDA 21,X
0206 F0 BEQ 0209
0208 C8 INY
0209 E8 INX
020A E4 CPX 10
020C D0 BNE 0202
020E 84 STY 20
0210 4C JMP E1A1
```

Figure 5-2. Symbolic Code for Zeroes-Counting Program

```
       LDX  #00      ;POINTER = START OF TABLE
LOADE  LDY  #00      ;NUMBER OF ZEROES = 0
       LDA  21,X     ;FETCH ELEMENT
       BEQ  NEXT     ;THIS ELEMENT = 0?
       INY           ;YES, ADD 1 TO ZEROES COUNT
NEXT   INX           ;POINT TO NEXT ELEMENT
       CPX  10       ;ANY MORE ELEMENTS?
       BNE  LOADE
       STY  20       ;NO, SAVE NUMBER OF ZEROES
       JMP  E1A1     ;  AND RETURN TO MONITOR
```

Figure 5-3. Mnemonic Entry Code for Zeroes-Counting Program

4. After entering the program into memory, reinitialize the Program Counter to $0200 and execute the program with the G/. command.

5. Now check the result by displaying the zeroes-count in location $20. Since our ten-element data table (locations $21 through $2A) contained all zeroes, location $20 should contain $0A. But, instead of $0A, you discover that

   ($20) = 01

   This result implies that the program executed the INY instruction only once. To find out when INY was executed, we will assign a breakpoint to this instruction's address, $0208, using the B command. To do this:

   - Press the $ key, to clear all breakpoints

- Press the B Key, to set the breakpoint, and enter 0=0208 in response to the prompt BRK/.
- Press the 4 key, to enable the breakpoint feature

6. With the breakpoint assigned and enabled, run the program once more, from the beginning. The AIM 65 should display

```
0208  C8  INY
```

At this point, the R6502 CPU has halted prior to executing the INY instruction. How many data table values did it check before hitting the INY instruction? The answer lies in the contents of the X Register, which holds the table index.

7. Press the R key, to get a register display. At this point, you should find out that

(X) = 0A

What does this value of X indicate? It indicates that the last location to be loaded into the Accumulator by the LDA 21,X instruction was location $2B (since $21 + $0A = $2B). Apparently, when the Branch instruction BEQ 0209 tested this location (now in the Accumulator), the test "failed" and program execution will continue with the INY instruction. Since location $2B contains a non-zero value, the BEQ 0209 instruction is clearly erroneous. It has been causing a branch past INY when a zero value, rather than a non-zero value, is loaded into the Accumulator. The correct branch instruction is:

```
BNE  0209
```

Change BEQ 0209 to BNE 0209, disable the breakpoint feature (with the 4 key) and execute the program again, from the beginning.

What does location $20 contain now? A check should show:

($20) = 00

Location $20 still holds no zeroes count! What can be wrong now? Is it possible that the INY instruction is still being bypassed, even though the branch instruction has been corrected? We can find out by running the program with both the instruction trace and the register trace turned on. This will produce a print-out of each instruction as it is executed, along with the register values resulting from that instruction.

8. Press Z to activate the instruction trace and V to activate the register trace, then execute the program from the beginning again.

Press ESC after a few passes through the loop, to terminate the print-out. Figure 5-4 shows the type of print-out you will get.

9. The boxed portion of the print-out in Figure 5-4 indicates the problem area, because it encloses the spot where Y changes from 01 to 00. Note that Y is being re-initialized to 00 by the instruction LDY #00 at location $0202. How did the program get back to *that* instruction? The instruction BNE 0202 at location $020C caused a branch to it.

We have just isolated the second error in the program. This branch should go fetch the next table element (LDA 21,X), and should not re-initialize the Y Register. The correct branch instruction is, then,

BNE 0204

```
<*>=0200
<Z>ON
<V>ON
<R>
 **** PS AA XX YY SS
 0200 00 00 00 00 FF
<G>/.
 0202 22 00 00 00 FF
 0202 A0 LDY #00
 0204 22 00 00 00 FF
 0204 B5 LDA 21,X
 0206 22 00 00 00 FF
 0206 D0 BNE 0209
 0208 22 00 00 00 FF
 0208 C8 INY
 0209 20 00 00 01 FF
 0209 E8 INX
 020A 20 00 01 01 FF
 020A E4 CPX 10
 020C A0 00 01 01 FF
 020C D0 BNE 0202
 0202 A0 00 01 01 FF
 0202 A0 LDY #00
 0204 22 00 01 00 FF
 0204 B5 LDA 21,X
 0206 22 00 01 00 FF
 0206 D0 BNE 0209
 0208 22 00 01 00 FF
 0208 C8 INY
 0209 20 00 01 01 FF
 0209 E8 INX
 020A 20 00 02 01 FF
 020A E4 CPX 10
 020A E4 CPX 10
```

Figure 5-4. Print-out With Instruction and Register Traces

10. Change BNE 0202 to BNE 0204, then press Z and V (to deactivate the traces) and run the program once more from the beginning.

11. What does location $20 contain after this new run through the program? You should discover that:

    ($20) = 0B (or something other than 0A)

Apparently, we still have problems. Location $20 should properly contain 0A, since locations $21 through $2A contain a total of 10 (hex 0A) zeroes. Evidently, location $20 contains a value greater than 0A because the program has somehow processed more than 10 locations in memory -- and some of these extra locations contained zero!

12. We can find out how many locations the program actually examined by looking at the final contents of the X Register, which contains the element pointer. At this point,

    (X) = 21

    That is, the program has actually processed 33 locations, rather than the intended 10 locations!

13. To find out why so many locations were checked, we must examine the portion of the program that stops the location-checking operation. This function is provided by the instructions CPX 10 and BNE 0204, at locations $020A and $020C, respectively. The culprit is obvious: it is the CPX 10 instruction. This instruction is intended to compare the contents of the X Register to the value 10. The instruction should be CPX #10, since we want to compare X to the number 10, not to the contents of memory location 10, Furthermore,we want the decimal number 10, not the hexadecimal number 10 (remember, AIM 65 assumes that the operand represents a hexadecimal value). The instruction should be

    CPX #0A

14. Change CPX 10 to CPX #0A, then execute the program once more from the beginning. At long last, the results are:

    (X) = 0A

    ($20) = 0A

15. Test the program with some other data that is not all zeroes, to convince yourself that the program is now correct.

The final program is shown in Figure 5-5.

```
0000      *=0200
0200 A2 LDX #00
0202 A0 LDY #00
0204 B5 LDA 21,X
0206 D0 BNE 0209
0208 C8 INY
0209 E8 INX
020A E0 CPX #0A
020C D0 BNE 0204
020E 84 STY 20
0210 4C JMP E1A1
```

Figure 5-5. Final Zeroes-Counting Program

SUMMARY

Common programming errors to look for are:

1. Forgetting to initialize variables. Don't assume anything is zero when you start.
2. Confusing data and addresses. The contents of memory location $10 could be anything, including (but not necessarily) the number $10.
3. Branching on the wrong condition; e.g., branching on not equal instead of equal.
4. Confusing decimal and hexadecimal numbers. Decimal 10 is hex 0A; hex 10 is decimal 16.
5. Accidentally re-initializing a register or memory location, by branching to the wrong place.
6. Incorrect keyboard entries. You should examine the entire program (by disassembling it, with the K command) before you run it.
7. Forgetting to update a counter or pointer. Watch for loop controls that must be updated regardless of which path the program follows.

6

MULTIPLICATION OPERATIONS, WITH SHIFT & ROTATE

OBJECT

To become familiar with the shift and rotate instructions, and to see how they apply to multiplication.

PRE-LAB PREPARATION

Read Chapter 10 of the R6500 Programming Manual.

DISCUSSION

Shift and Rotate Instructions

The R6502 has four instructions that cause the 8-bit contents of an operand (a memory location of the Accumulator) to be displaced by one bit position to the left or to the right. Two of these instructions "shift" the operand, the other two instructions "rotate" the operand.

For all four instructions, the Carry bit of the Processor Status Register acts as a "ninth bit" extension of the operand; that is, Carry receives the value of the bit that is displaced out of one end of the Accumulator or memory location (Bit 0 for a right shift, Bit 7 for a left shift). In a shift operation, the vacated bit position at the opposite end of the operand (Bit 7 for a right shift, Bit 0 for a left shift) is reset to 0. In a rotate operation, the vacated bit position of the operand receives the initial (unrotated) value of the Carry bit.

These are the shift and rotate instructions:

INSTRUCTION	DESCRIPTION
ASL	Shift Left One Bit
LSR	Shift Right One Bit
ROL	Rotate Left One Bit
ROR	Rotate Right One Bit

The operations of these instructions are illustrated in Figure 6-1.

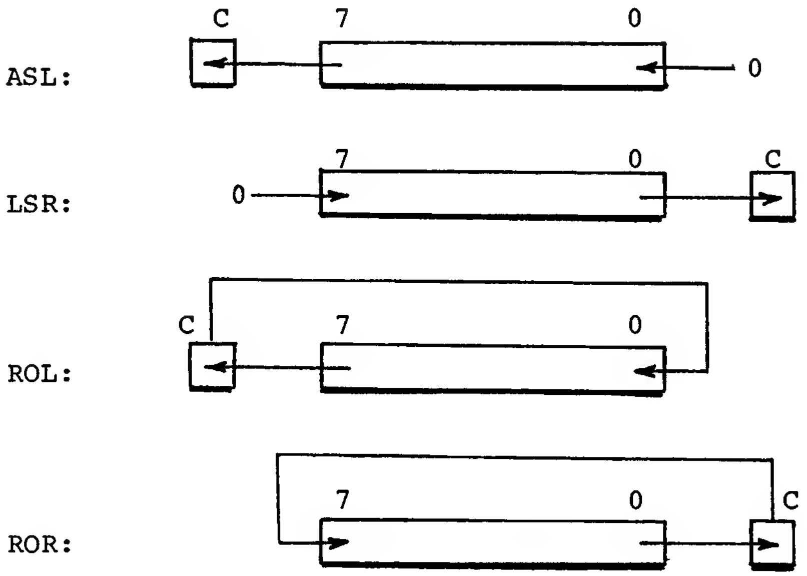


Figure 6-1. Shift and Rotate Instructions

As mentioned previously, the shift and rotate instructions can operate on either the Accumulator or a location in memory. To operate on the Accumulator, you simply put an A in the operand field of the instruction, like this:

```
ASL   A
```

The shift and rotate instructions are the only instructions that use the Accumulator addressing mode.

In addition to their affect on the Carry flag, the shift and rotate instructions affect two other flags in the Processor Status Register:

- ASL, ROL and ROR cause the Negative (N) flag to be set if Bit 7 of the shifted result is a logic 1; otherwise N is reset. The LSR instruction always resets the N flag, since it shifts a 0 into Bit 7.
- The Zero (Z) flag is set if the shifted result is zero; otherwise it is reset.

To show how these instructions work, consider an operand that contains a hexadecimal 34 (binary 00110100, decimal 52) and the Carry flag is set to 1. Here is how the operand and Carry flag would be altered for each of our four shift and rotate instructions:

CARRY FLAG	BIT POSITION								
	7	6	5	4	3	2	1	0	
1	0	0	1	1	0	1	0	0	Before shift (= hex 34, dec 52)
0	0	1	1	0	1	0	0	0	After ASL (= hex 68, dec 104)
0	0	0	0	1	1	0	1	0	After LSR (= hex 1A, dec 26)
0	0	1	1	0	1	0	0	1	After ROL (= hex 69, dec 105)
0	1	0	0	1	1	0	1	0	After ROR (= hex 9A, dec 154)

Multiplication Operations

We will be using the shift and rotate instructions to multiply binary numbers, but before we do, let us review the mechanics of decimal multiplication, as we have always done it with pencil and paper. As you know, you write the multiplicand on a piece of paper, with the multiplier below it, and perform a series of partial multiplications, one for each digit in the multiplier. Each partial product is written directly below its multiplier digit, causing this product to be displaced one digit position to the left of the preceding partial product. When all partial products have been calculated, they are added to produce the final product.

For example, the multiplication of 124 by 103 looks like this:

124	Multiplicand
x 103	Multiplicand
372	Partial Product #1
000	Partial Product #2
124	Partial Product #3
12772	Final Product

In practice you normally do not write down the "all-zeroes" partial product, but it is shown here to emphasize one essential principle of multiplication: A zero multiplier digit causes a skip to the next digit position, without producing a partial product.

Remember, from fourth grade arithmetic, that each partial product is displaced one digit to the left of its predecessor to reflect the higher decimal weight of its multiplier digit. That is, moving from right to left in the multiplier, each digit has a weight that is ten times greater than the digit to its right. For this reason, the preceding example could be written in this form:

$$103 \times 124 = (3 \times 124) + (0 \times 124) + (100 \times 124)$$

or, better yet, in this form:

$$103 \times 124 = (3 \times 10^0 \times 124) + (0 \times 10^1 \times 124) + (1 \times 10^2 \times 124)$$

With this groundwork, let us discuss binary multiplication. Binary multiplication is much simpler than decimal multiplication, because binary multipliers consist only of the digits 1 and 0. Therefore, the partial product is always a copy of the multiplicand (multiplier digit = 1) or zero (multiplier digit = 0).

The binary equivalent of our 103 x 124 example looke like this:

```
       01111100    Multiplicand (= 124)
     x 01100111    Multiplier (= 103)
       01111100
      01111100
     01111100
    00000000
   00000000
  01111100
 01111100
00000000
011000111100100    Final product (= 12772)
```

How can such a multiplication be performed with a computer? Well, like most 8-bit microprocessors, the R6502 has no multiply instruction, so the multiplication must be performed as we just did it: with a series of additions. However, there are some important differences between the pencil-and-paper approach and its computer counterpart. When multiplying by hand, you calculate all of the partial products, then add them to generate the final product. With a computer program, instead of waiting until all partial products are derived before generating the final product, it is much more efficient to update the partial product after processing each multiplier bit. By maintaining this running total, the final product is available when the highest bit of the multiplier has been processed.

Just as the significance of digits in a decimal (or base 10) number increases by a factor of ten, the significance of digits in a binary (or base 2) number increases by a factor of two, as you progress from right to left. So each binary digit is two times more significant than the digit on its right. As discussed previously, this increasing significance causes each partial product to be displaced to the left before writing it down on paper. With a computer program, it is easier to rotate the current partial product one bit position to the right, thereby aligning it to receive the next multiplier bit's contribution.

How long will the final product be? Well, our 103 x 124 binary multiplication had an 8-bit multiplicand, and yielded a 15-bit final product. Since two bytes in memory will be needed to hold this 15-bit product, we can observe that a one-byte multiplier and a one-byte multiplicand yielded a two-byte product. This was not coincidental; the length of the product will always be as long as the combined lengths of the multiplier and multiplicand. (An "n"-byte multiplier and an "m"-byte multiplicand will produce an "n + m"-byte product!)

Figure 6-2 illustrates what we have learned about multiplication thus far, by showing the sequence for processing one bit in the multiplier. Note that the state of the multiplier, shifted into Carry, determines which of two possible paths are followed:

- If Carry = 1, the multiplicand must be added to the high-order byte of the partial product before this product is rotated.
- If Carry = 0, the partial product is rotated to the right, without an addition.

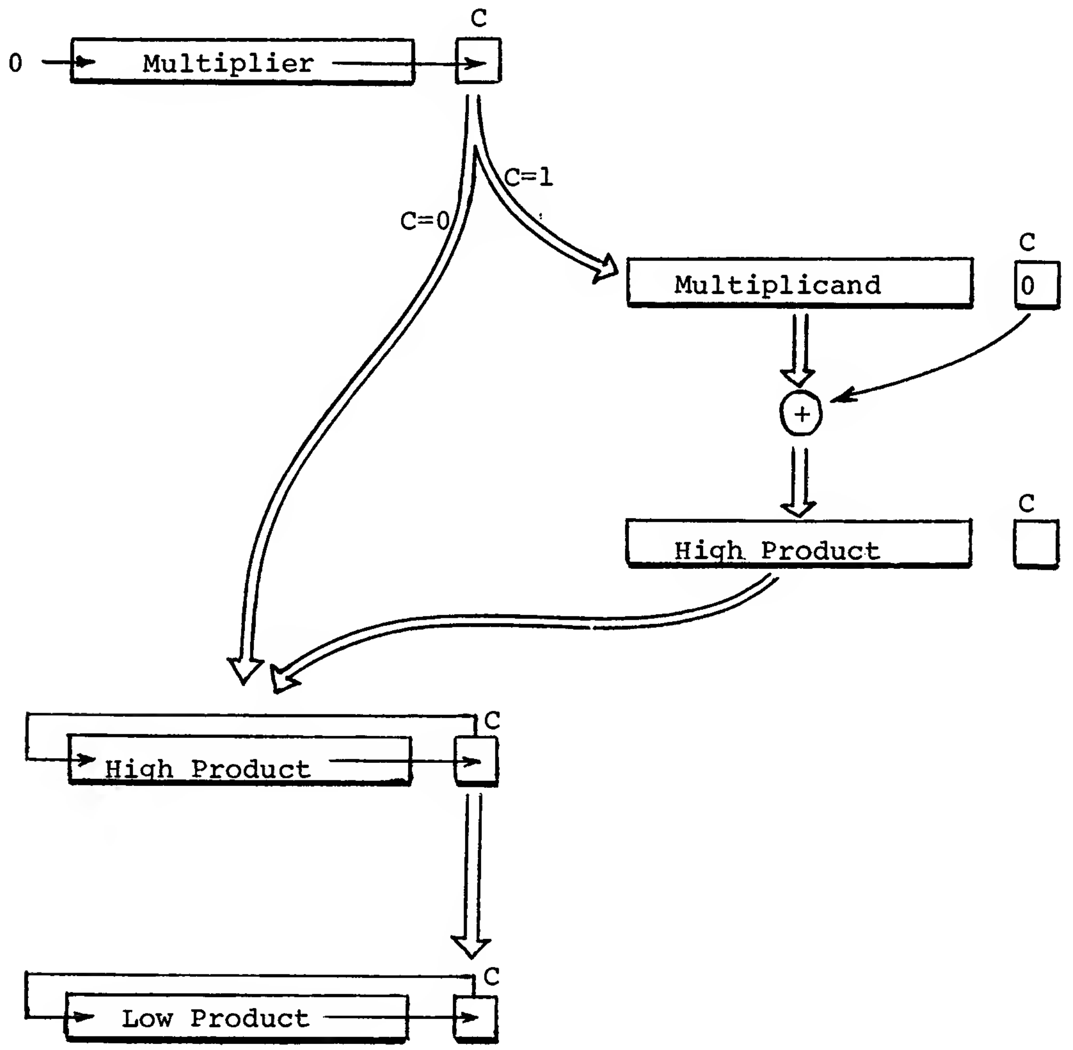

C
0
Multiplier
C=1
C=0
C
Multiplicand
0
+
C
High Product
C
High Product
C
Low Product

Figure 6-2. Processing a Multiplier Bit

Programming Multiplication Operations

Multiplication programs should follow this general sequence of operations:

1. Initialize the product to zero.

2. Shift the multiplier right, into Carry, using the LSR instruction.

3. If Carry = 1, add the multiplicand to the high-order byte of the partial product. If Carry = 0, skip to Step 4.

4. Rotate the partial product right, into Carry, using two ROR instructions.

5. Repeat Steps 2 through 4 for each bit in the multiplier.

To multiply multiprecision numbers, the most-significant byte of the multiplier will be right-shifted with an LSR instruction. Less-significant bytes must be right-rotated with ROR instructions.

PROCEDURE

1. In the discussion of shift and rotate instructions at the beginning of this experiment, we showed the effect of the ASL, LSR, ROL and ROR instructions on the binary pattern 00110100, with the Carry bit set to 1. Verify the results shown in the discussion by executing four, 3-instruction programs of the form

   ```
   LDA  #34
   SEC
   nnn  A
   ```

   where nnn represents the mnemonics ASL, ROL and ROR in the four programs.

2. Write a program to multiply the 8-bit unsigned integer multiplicand in location \$21 by an 8-bit unsigned multiplier in location \$20, storing the 16-bit product in locations \$22 (low byte) and \$23 (high byte).

   Draw the flowchart below.

Write the program code below.

3. Run your program for the four cases listed below, and complete the table.

Multiplier $20		Multiplicand $21		Product $22,$23	
Dec.	Hex.	Dec.	Hex.	Dec.	Hex.
255	FF	1	01	255	
103	67	124	7C	12772	
255	FF	0	00	0	
255	FF	255	FF	65025	

7

DIVISION OPERATIONS

OBJECT

To learn how the shift and rotate instructions can be applied to division operations.

DISCUSSION

Division involves calculating the number of times one number (called the divisor) can be subtracted from another number (called the dividend). The integer value of the resulting number of subtractions is called the quotient, and the "leftover" fraction is called the remainder.

For example, dividing 1265 by 13 with pencil and paper would look like this:

```
      0097 ←——— Quotient
   13 ⌈1265
      -0
       12
      - 0
       126
      -117
         95
        -91
          4 ←——— Remainder
```

Although it is almost second-nature by now, in this pencil-and-paper method we actually obtain the quotient digits by recalling the "division tables" that were hammered into us in elementary school. As with multiplication, this method cannot be applied to a computer solution, because the R6502--like most 8-bit microprocessors--has no divide instruction. In Experiment 6, multiplication was performed with a series of add operations. Division, similarly, can be performed with a series of subtraction operations.

Specifically, division can be performed with a series of trial subtractions, using the R-6502's Compare instructions, which were covered in Experiment 4. In these trial subtractions, the divisor is compared with the partial dividend, beginning with the left-most (most-significant) digit of the dividend. With this approach, the logical steps for the preceding divide operation are as follows:

(1) How many times can 13 be subtracted from 1? The answer is 0, so enter 0 in the quotient and bring down the next digit, 2, to form 12.

(2) How many times can 13 be subtracted from 12? The answer is again 0, so enter another zero in the quotient and bring down the next digit, 6, to form 126.

(3) How many times can 13 be subtracted from 126? The answer is 9, so enter 9 in the quotient and subtract 117 (9 times 13) from 126.

(4) The subtraction leaves a remainder of 9, so bring down the final digit of the dividend, 5, to form 95.

(5) How many times can 13 be subtracted from 95? The answer is 7, so enter 7 in the quotient and subtract 91 (7 times 13) from 95.

(6) All of the digits of the dividend have been tested, so the division is complete. The quotient is 97 (ignoring leading zeroes) and the remainder is 4.

Dividing binary numbers is similar to dividing decimal numbers, but is much easier. With binary numbers, your quotient digits will always be either 0 or 1. If the divisor can be subtracted from the partial dividend, the quotent digit will be a 1, otherwise it will be a 0.

The binary equivalent of our divide-1265-by-13 example is shown in Figure 7-1. As with decimal division, a new quotient digit (1 or 0 for the binary version) is always entered to the right of the previously-entered quotient digits. In a computer program, this is accomplished by shifting the old partial quotient to the left and entering a new quotient digit into the vacated least-significant bit position.

The computer implementation requires the dividend to be left-shifted, too, to form a partial dividend to which the divisor is compared (and, if possible, subtracted).

The fundamental operations for dividing binary numbers with a computer are:

(1) Shift the quotient (initially zero) left, to provide a (least-significant) bit position for the next quotient digit.

(2) Shift the dividend left, displacing its most-significant digit into Carry.

(3) Rotate the partial dividend left, bringing in the new dividend digit from Carry.

(4) Compare the divisor to the partial dividend.

(5) If the divisor is less than or equal to the partial dividend, subtract the divisor from the partial dividend and enter a 1 in the quotient.

(6) If any digits remain in the dividend, go back to Step 1.

```
     00001100001
1101|10011110001
    -0
     10
    - 0
     100
    -  0
     1001
    -   0
     10011
    - 1101
       1101
      -1101
          01
        - 0
          010
        -  0
          0100
        -   0
          01000
        -    0
          010000
        -  1101
             100
```

Figure 7-1. Binary Division

The preceding procedure can be shortened from six steps to five steps based on this observation: the dividend and quotient can share the same space in memory! How is this possible? The answer is simple: the dividend and quotient can share the same space in memory because, at any given time, the length of the quotient is equal to the number of digits that have been left-shifted out of the dividend. Never do you need more quotient digit positions than are available with the current dividend.

In the memory they share, the dividend will occupy the high-order bit positions and the quotient will occupy the remaining, low-order bit positions. So, as you can see, Steps 1 and 2 of our six-step divide procedure can be combined into a single step:

(1,2) Shift the dividend/quotient left, displacing the most-significant digit of the dividend into Carry.

Figure 7-2 shows this step, and the partial dividend rotate (Step 3).

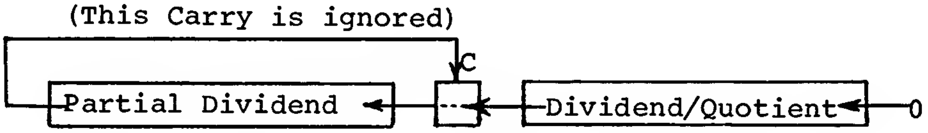


Figure 7-2. How Digits are Entered into the Partial Dividend

PROCEDURE

1. Write a program to divide the 8-bit unsigned integer dividend in location $21 by an 8-bit unsigned divisor in location $20. The 8-bit quotient should be returned in location $21, replacing the dividend, and the 8-bit remainder should be returned in location $22.

   Draw the flowchart below.

Write the program code below.

2. Run your program for the four cases listed below, and complete the table.

Divisor $20		Dividend $21		Quotient $21		Remainder $22	
Dec.	Hex.	Dec.	Hex.	Dec.	Hex.	Dec.	Hex.
1	01	255	FF	255		0	
15	0F	170	AA	11		5	
255	FF	1	01	0		255	
254	FE	255	FF	1		1	

3. As you know, dividing by zero produces an infinite quotient, and is therefore illegal. What instructions can be added to the beginning of your program to check for a zero divisor, to bypass the divide operation if the divisor is zero? List the instructions below.

   Add these instructions to your program, and run a test case to verify that they are correct.

8

SUBROUTINES AND THE STACK

OBJECT

To learn how to program using subroutines, and the use of the stack in this and other applications.

PRE-LAB PREPARATION

Read Chapter 8 in the R6500 Programming Manual and Section 6.9 in the AIM 65 User's Guide.

DISCUSSION

In 6500-based systems, the memory locations at addresses $0100 through $01FF have special meaning to the programmer insofar as these locations act as a stack. The stack functions as a temporary depository for two types of information: 1) register contents and 2) interrupt or subroutine return addresses.

Information is entered onto, and extracted from, the stack the same way we stack dishes in the kitchen. The last item to be placed on the stack will be the first item to be removed from it. For this reason, the R6502's stack is of a type that is usually referred to a "last-in, first out." As register contents or return addresses are stored on the stack, they are deposited in read/write memory at lower and lower addresses; the stack "builds" toward address 0.

The R6502 contains an eight-bit register that maintains the address of the next "free" stack location. This register, the Stack Pointer, is automatically decremented by one after an information byte is pushed onto the stack, and is automatically incremented by one before an information byte is pulled off the stack. Because all stack addresses are of the form $01XX, the Stack Pointer holds only the low-order eight bits of the address; the high-order eight address bits ($01) are automatically added on by the R6502 at the time of a stack operation. For example, if the Stack Pointer contains $A3, the next byte of information will be pushed onto the stack at location $01A3, or pulled from the stack at location $01A4.

For all practical purposes, you can consider the operation of the stack as being automatic, and you should not really concern yourself with which stack location is being accessed at any given time. There are, however, two important rules for programming stack operations:

1. Every byte of information that is pushed onto the stack must eventually be pulled from the stack, and
2. Information is retrieved from the stack in the reverse order from which it was entered onto the stack.

Saving Register Contents on the Stack

The R6502 has instructions that permit the contents of both the Accumulator and the Processor Status Register to be saved on the stack. They are:

INSTRUCTION	DESCRIPTION
PHA	Push Accumulator on Stack
PHP	Push Processor Status on Stack
PLA	Pull Accumulator from Stack
PLP	Pull Processor Status from Stack

Aside from the fact that they push different registers onto the stack, the PHA and PHP instructions work identically. In each case, the contents of the register are pushed onto the stack at the location being pointed to by the Stack Pointer, then the Stack Pointer is decremented by one, to the next lower address.

Similarly, the PLA and PLP instructions increment the Stack Pointer to the next higher address, then load the contents of the addressed memory location into the appropriate destination register.

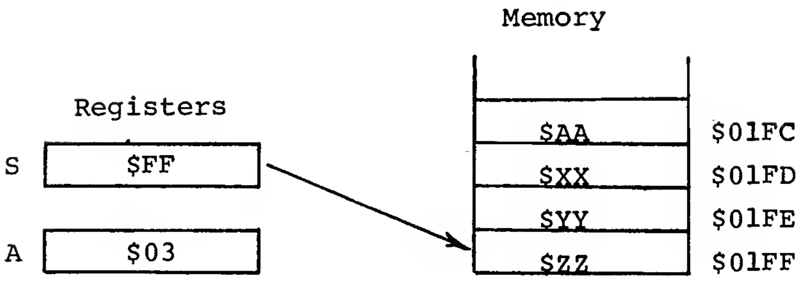
Figure 8-1 shows the effect of the PHA instruction on the stack. In the upper drawing, the Stack Pointer (S) is pointing to the top of the stack, location $01FF, and the Accumulator contains $03. After PHA is executed (lower drawing), the Stack Pointer has been decremented to $FE and the contents of the Accumulator have been stored in location $01FF. If we were to execute a PLA instruction after the push, the Stack Pointer would be incremented, to point to $01FF, and the Accumulator would be loaded with the contents of location $01FF.

Why would someone want to save register contents on the stack? There is really only one reason why these values would be saved: to preserve their contents while the registers are being manipulated by other programming operations. This is particularly true in subroutines, but is also applicable to routines in which registers are serving two functions, or where the final contents of a register are unpredictable.

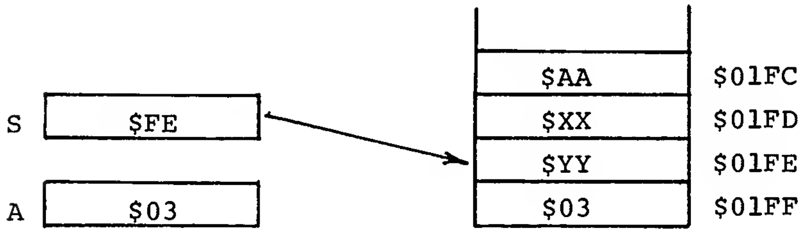
With only four instructions, it looks as if only the Accumulator and the Processor Status Register can be saved on the stack. That is true as far as the instructions go, but using the R6502's register transfer instructions, we can move the X and Y Registers into the Accumulator, push them onto the stack, and later pull them off and restore them with additional register transfer instructions. The register transfer instructions are:

INSTRUCTION	DESCRIPTION
TAX	Transfer Accumulator to Index X
TAY	Transfer Accumulator to Index Y

TSX	Transfer Stack Pointer to Index X
TXA	Transfer Index X to Accumulator
TXS	Transfer Index X to Stack Pointer
TYA	Transfer Index Y to Accumulator



Before PHA



After PHA

Figure 8-1. How a Register is Pushed Onto the Stack

Subroutines

Many programs require a certain operation to be performed at several points in the program. To do this, you could certainly rewrite the entire series of required instructions in each place they are used, but this would be not only frustrating and time-consuming for the programmer, but it would also make programs much longer than they would be if this repetition could be avoided.

Well, as a matter of fact, all microprocessors do eliminate this need for recoding, by permitting the repeated code to be defined as a subroutine. A subroutine is a sequence of instructions that is written just once, but can be executed as needed from any point in the program. The process of transferring control to a subroutine is given the term calling. Therefore, subroutines are called. Once called, the R6502 executes the sequence of instructions in the subroutine, then returns control to the calling program.

From the preceding description, it is apparent that instructions which cause subroutines to be called must perform three functions:

1. They must include some provision for saving the contents of the Program Counter. Once the subroutine has been completed, this address will be used to return to the program.
2. They must cause the microprocessor to begin executing the subroutine.
3. They must use the stored contents of the Program Counter to return to the program, and continue executing the program at this point.

These three functions are performed by two R6502 instructions:

INSTRUCTION	DESCRIPTION
JSR	Jump to Subroutine
RTS	Return from Subroutine

The JSR instruction performs the return address-storing and begin-executing functions (1 and 2). The return address stored is the address of the third byte of the JSR instruction. (This address will be incremented by one, by the RTS instruction, to provide the proper destination.) The return address is stored on the stack, which means that the JSR instruction operates like the push instructions (PHA and PHP) that we described earlier in this experiment. Unlike PHA and PHP, which pushed one byte of data onto the stack, JSR pushes two bytes onto the stack -- the two-byte address in the Program Counter.

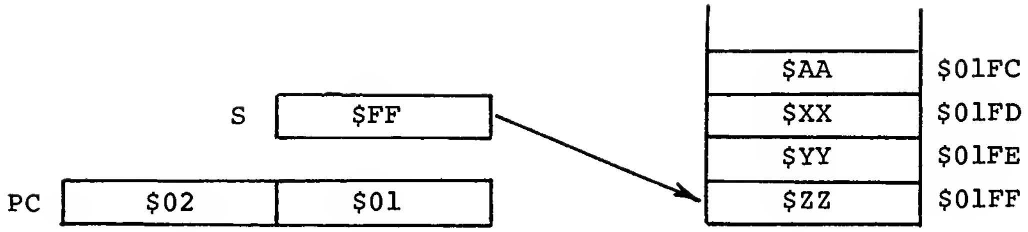
After storing the Program Counter contents on the stack, the JSR instruction loads the Program Counter with the absolute address from its second and third bytes, which transfers control to the starting address of the subroutine.

Let us consider a typical Jump to Subroutine instruction, JSR $0503, located in memory at locations $0201, $0202 and $0203.

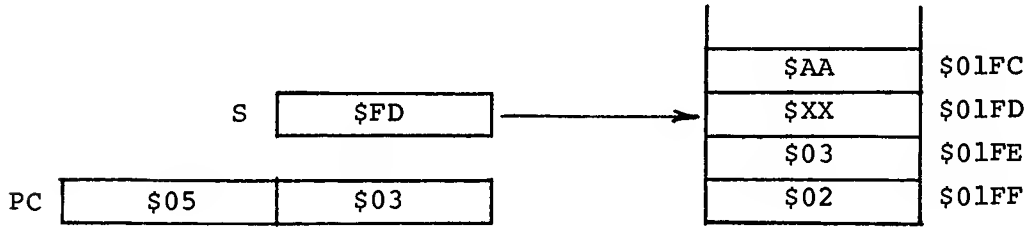
That is:

MEMORY LOC.	INSTRUCTION
$0201	JSR $0503
.	.
.	.
.	.
$0503	(First subroutine instruction)

Figure 8-2 shows the contents of the Stack Pointer (S), the Program Counter (PC) and the stack before and after the instruction JSR $0503 is executed. In the upper drawing, the Program Counter is pointing to the first byte of the JSR instruction ($0201) and the Stack Pointer is pointing to the next free location on the stack ($01FF is assumed). In the lower drawing, after the JSR $0503 instruction has been executed, the Program Counter is pointing to the starting address, or entry address, of the subroutine ($0503), the Stack Pointer is pointing to a new "next free stack location" ($01FD), and the two top bytes on the stack are the most-significant and least-significant address bytes of the JSR instruction's third bytes.



Before Executing JSR $0503

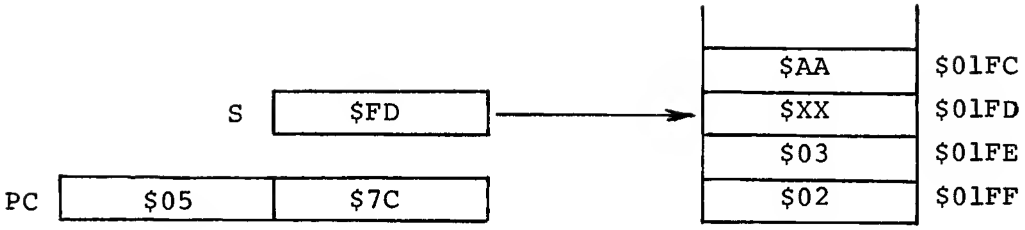


After Executing JSR $0503

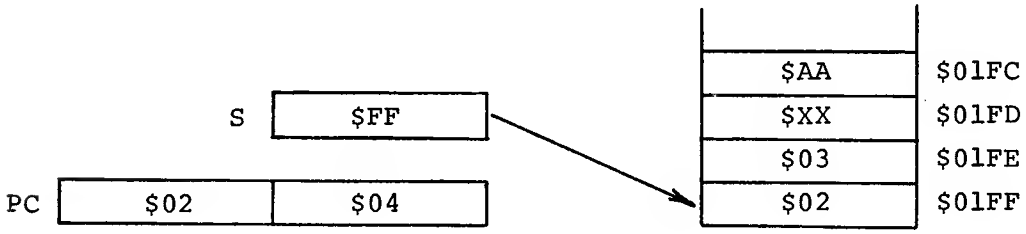
Figure 8-2. How JSR Saves a Return Address on the Stack

The Return from Subroutine (RTS) instruction causes the R6502 to return from the subroutine to the calling program (the program that contains the JSR instruction). In doing this, the RTS instruction causes the R6502 to continue program execution at an absolute address that is one greater than the address on the last two bytes of the stack. Recall that before loading the subroutine address into the Program Counter, the JSR instruction pushed the address of its third byte onto the stack. To retrieve that address, the RTS instruction increments the Stack Pointer, loads the low-order address byte into the lower half of the Program Counter, then increments the Stack Pointer again and loads the high-order address byte into the upper half of the Program Counter. The Program Counter is then incremented so that it addresses the instruction that follows JSR.

Recall our previous example in Figure 8-2, which illustrated stack operation due to executing the instruction JSR $0503. If the called subroutine contained an RTS instruction at location $057C, Figure 8-3 shows the contents of the Stack Pointer (S), the Program Counter (PC) and the stack before and after that RTS instruction is executed.


S
$FD
PC
$05
$7C
$AA
$XX
$03
$02
$01FC
$01FD
$01FE
$01FF

Before Executing RTS


S
$FF
PC
$02
$04
$AA
$XX
$03
$02
$01FC
$01FD
$01FE
$01FF

After Executing RTS

Figure 8-3. How RTS Pulls an Address from the Stack

PROCEDURE

1. Write a program to do the following:

   A. Initialize the Stack Pointer to "point to" location $01FF.

   B. Load the Accumulator, X Register and Y Register with the values $00, $01 and $02, respectively.

   C. Push A, X and Y onto the stack.

   Write the code for this program below. The final instruction should be BRK.

2. To see the effect of your program on the stack, use the M command to examine the initial contents of locations $01FB through $01FF, and enter your observations in the BEFORE block below. After doing that, set the AIM 65's RUN/STEP Switch to RUN, enter the program into AIM 65 (starting at location $0200), execute it, and examine locations $01FB through $01FF again. Enter these new observations in the AFTER block below.

BEFORE	Location	AFTER
	$01FB	
	$01FC	
	$01FD	
	$01FE	
	$01FF	

What address is now in the Stack Pointer?

Stack Pointer =

If you were now to initiate a PLA instruction (Pull Accumulator From Stack), what would you expect to find in the Accumulator?

After PLA, Accumulator =

3. Add a PLA instruction to your program, then run the modified program. What value has been loaded into the Accumulator?

   Accumulator =

   Does this agree with your answer to the last question in Step 2? If not, go back and correct your program. (Make sure the RUN/STEP switch was set to RUN when you executed the program!)

4. To find out how subroutines operate, you are to develop a subroutine which adds 5 to the contents of location $20. The subroutine instructions should start at location $0300 in memory. Write the code for the subroutine below.

   Now, write a program that initializes location $20 to zero, then calls your subroutine three consecutive times (which should yield a value of 15, or $0F, in location $20). The calling program should start at location $0200. Write the code for this program below.

5. Enter the calling program and the subroutine into AIM 65 memory, then execute the program. Record the result below.

   Location $20 =

6. To what value does AIM 65 initialize the Stack Pointer at power-on? Find out by turning the power off, then on, and displaying the Stack Pointer value.

   At power-on, Stack Pointer =

9

UNORDERED LISTS

OBJECT

To learn the fundamentals of processing unordered data.

DISCUSSION

There are many ways in which information in memory can be organized for processing. These organizational techniques vary with the application, and are categorized with such names as lists, arrays, strings, look-up tables and vectors. As expected, the subject can (and does) fill many volumes, but in this manual we will concentrate on one of the most basic types of organization, lists.

Lists consist of units of data (one or more bytes), called elements, arranged sequentially in memory. The sequence can be consecutive, in which each element occupies one or more adjacent memory locations, or linked, in which each data element is followed by an address pointer to the next element in the list. Further, the elements can be arranged randomly, or in ascending or descending order. This experiment will deal with unordered lists. The next experiment, Experiment 10, will discuss how to arrange the list's elements in order.

Unordered Lists

In our ordered society, where telephone book listings are arranged alphabetically and house numbers increase (or decrease) systematically as you go up or down a street, unordered anythings seem somehow inferior to us. However, unordered lists of data are a fact of life in many applications, and represent a common way to store random, chronologically-derived, or dynamically changing data (especially data from an experiment).

Typically, unordered lists are comprised of an element count byte (or bytes) and one or more data elements. There are three primary operations that can be performed on a list: adding an element to the list, deleting an element from the list, and searching the list for a certain element value.

For a list whose elements are stored in consecutive memory locations, an element is added by storing it at the end of the list, then adding 1 to the element count. An element is deleted by simply moving the remainder of the list (all elements following the one to be deleted) upward -- toward address 0 -- by one element, then subtracting 1 from the element count. Since the list is unordered, the search operation's procedure is self-evident: each element in the list, starting with the first element, must be compared to the search value.

The search continues until a match is found or you reach the end of the list.

The compare portion of a search procedure follows the same rules we learned in Experiments 4 and 8: single-byte (8-bit) numbers can be compared with one of the 6502's three compare instructions (CMP, CPX or CPY), but multi-byte comparisons require a compare instruction to compare the least-significant bytes and a subtract instruction (SBC) to compare all remaining, more-significant bytes.

PROCEDURE

1. In this experiment you are to write a program that searches an unordered list in memory, to find out whether or not the list contains the value being held in location $2F. If the search value is found in the list, the element number of the matching element (1, 2, 3, etc.) is returned to the Y Register. If the search value is not found in the list, the Y Register contains zero upon return.

   The starting address of the list is in locations $30 (low address byte) and $31 (high address byte). The length of the list is contained in the list's first byte. See Figure 9-1.

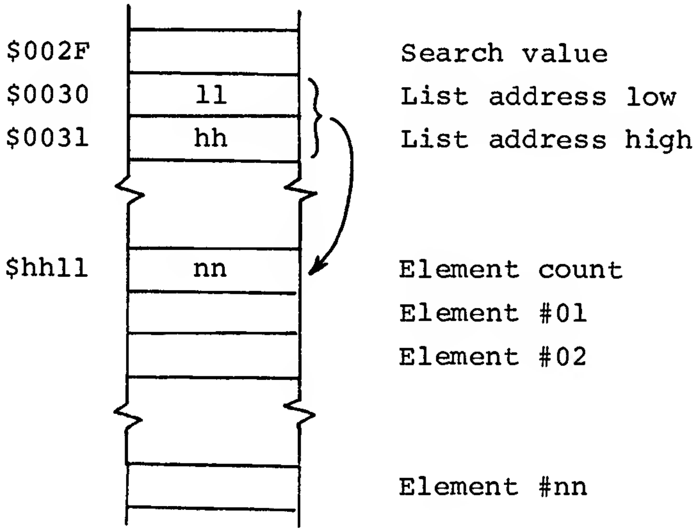

$002F
$0030
$0031
$hhll
ll
hh
nn
Search value
List address low
List address high
Element count
Element #01
Element #02
Element #nn

Figure 9-1. Unordered List in Memory

Flowchart the problem in the space below. The easiest solution involves using the X Register to hold the element count from the list's first byte (decrement X as you advance through the list), and using the Y Register to hold the index to elements in the list (increment Y as you advance through the list).

2. Write the program code below, then enter it into AIM 65's memory, starting at location $0200. Note that because the list's starting address is contained in Page 0 of memory, your program must use Indirect Indexed addressing -- with operands of the form (30),Y -- to access the list.

3. Assume that you wish to search for the value $55 in a list that occupies locations $40 through $4A. Store the following list values in memory:

LOCATION	CONTENTS
$40	$0A
$41	$A2
$42	$00
$43	$40
$44	$40
$45	$CE
$46	$5A
$47	$FA
$48	$55
$49	$06
$4A	$4F

Store the appropriate initial values of locations $2F, $30 and $31 in memory, and record your entered values below.

Search Value	($2F)	=
List Address, Low	($30)	=
List Address, High	($31)	=

Run your program for a search value of $55, and record the result indication below.

Y =

Since $55 is the eighth element in the list, the Y Register should contain the value $08. If it does not, go back and correct your program until you get this result.

4. The program you just developed does nothing more than find out whether or not the search value is contained in the list. Now, you are to modify this program so that if if finds the search value in the list, it will delete the matching element from the list.

   In the space below, draw a flowchart for the portion of the program that will delete an element from the list if a match occurs. It should consist of a loop in which you decrement the count and move one element up in the list, until the count goes to zero (at which point you must decrease the count value in the list's first byte location).

5. Write the code for the program that deletes an element from the list if a match occurs.

6. Using the list values from Step 3, run your delete program for the search value $55, and record the result indications below.

($40) =

($47) =

($48) =

($49) =

(4A) =

10

SORTING UNORDERED DATA

OBJECT

To learn the principles of sorting unordered data.

DISCUSSION

Although unordered data, such as we covered in Experiment 9, is perfectly acceptable for many applications (e.g., information that will be plotted versus time), it is often easier to process data that has been arranged in increasing or decreasing order. How can a list of unordered data be so arranged? One of the most common techniques is the technique known as bubble sorting.

Bubble sorting arranges data elements in increasing order, for example, by causing higher-valued elements to "rise" upward in memory (away from location 0), as soap bubbles rise upward into the sky. A bubble sort finds the relative magnitude of a data element by performing a series of comparisons. Here is how it is done: the list's first element is compared with the second element. If the first element is greater, the elements are exchanged. Then the second element is compared with the third element, exchanged if required, and so on. By the time the R6502 get to the end of the list (completes one "pass" through the list), the largest-valued element will have been "bubbled up" to the list's last element position.

The bubble sort procedure requires the R6502 to make another pass through the list after each pass that included an element exchange operation. The exchange/no-exchange indication is usually provided by a special exchange flag in the program.

Let us take a look at a simple example, to see how the bubble sort actually operates. Consider a five-element list that is initially arranged like this:

05 03 04 01 02

After the first pass through the list, the arrangement will be:

03 04 01 02 05

As you can see, the largest element in the list, 05, has "bubbled up" to the end of the list. The next pass will yield:

03 01 02 04 05

Now the 04 value has bubbled up to the next-to-last position in the list. The next pass will complete the ordering operation, giving:

01 02 03 04 05

However, since this pass included an exchange (in fact, it included two exchanges), the microprocessor will make one more pass through the list. This final pass will produce no exchanges, so the sort is finished.

The preceding example not only demonstrates how bubble sorting works, but also gives an indication of the type of performance you can expect from it. Note that four passes were required to sort a partially-ordered, five-element list. If the list was totally ordered at the outset, it would still take one pass for the R6502 to deduce this fact from the resulting status of the exchange flag. Conversely, if the list was initially arranged in descending order (the worst case), the bubble sort would require five passes to order the list.

The general rule is that the R6502 will have to make from one to N passes to bubble sort an N-element list. This means that an average of N/2 passes are required to sort an N-element list.

Processing Ordered Lists

How do the add, delete and search operations on an ordered list differ from those same operations on an unordered list? The delete operation is identical for both types of list: an element can be deleted by moving all subsequent elements up (toward location 0) by one position, thereby overlaying the deleted element. As you may have guessed, adding an element to an ordered list involves a similar operation -- move all higher-valued elements down (away from location 0) one position, insert the new element into the gap that has been created, and update the element count by one.

What about searching an ordered list? Can an ordered list be searched sequentially, element-by-element, as an unordered list is searched? Of course it can, but a sequential search on an ordered list makes as little sense as searching for a number in a phone directory, name-by-name, from the beginning. There are quite a few well-known search techniques for ordered lists -- and most will be faster and more efficient than the sequential search technique, for all but the shortest lists. Interested students are referred to any of the many books on programming techniques for more information on this subject. A few sources are:

1. Knuth, D.E. The Art of Computer Programming, Volume III: Sorting and Searching. Addison-Wesley Publishing Co., Reading, 1978. This book represents a classic treatment of the subject.
2. Scanlon, L.J. 6502 Software Design. Howard W. Sams & Co., Inc., Indianapolis, IN, 1980, pp. 102-107. Describes binary search technique.
3. Bentley, J.L. "An Introduction to Algorithm Design," Computer, February 1979, pp. 66-78.

PROCEDURE

1. Write a program that uses the bubble sort algorithm to arrange the 8-bit elements of a list in ascending order. The starting address of the list is in locations $30 (low address byte) and $31 (high address byte). The length of the list is contained in the list's first byte. Use location $32 to hold an exchange flag, an indicator that can be interrogated upon completion of a sorting pass to find out whether any elements were exchanged during that pass (flag = -1) or the pass was executed with no exchanges (flag = 0). Flowchart the problem in the space below.

Write the program code below, then enter it into AIM 65 memory, starting at location \$0200. As in Experiment 9, you must use Indirect Indexed addressing -- with operands of the form (30),Y -- to access the list.

Hint: The most difficult part of this problem will be in writing the instructions that exchange elements within memory. Since both X and Y are used in the program, they cannot be used for temporary storage during the exchange. As a solution, use the stack to hold the lower-addressed element while you move the higher-addressed element.

2. Store the following unordered list in memory.

LOCATION	CONTENTS
$40	$0A
$41	$A2
$42	$00
$43	$40
$44	$40
$45	$CE
$46	$5A
$47	$FA
$48	$55
$49	$06
$4A	$4F

Note that this is the same list we worked with in Step 3 of Experiment 9. The list has 10 elements (locations $41 through $4A) and an element count byte (location $40).

3. Order the list given in Step 2 using your program, and record the result below.

LOCATION	CONTENTS
$40	
$41	
$42	
$43	
$44	
$45	
$46	
$47	
$48	
$49	
$4A	

11

CODE CONVERSION FROM INPUT

OBJECT

To learn the techniques for converting input data to a form that is useable by the R6502 CPU.

DISCUSSION

In all computer systems, there must be a way of communicating digital information between the processor and external devices. That is, there must be a way to input data from a teletypewriter, card reader, cassette, floppy disk, or keyboard. There must also be a way to output data to a printer, display, card punch, cassette, floppy disk, or teletypewriter.

The data must, of course, be in a form that is recognizable to the recipient (the peripheral device or the R6502 microprocessor). Depending on its design, a peripheral device can operate on data in one of a number of forms, including ASCII (American Standard Code for Information Interchange), EBCDIC (Extended Binary-Coded Decimal Interchange Code), binary or Gray code. The keyboard, printer and display on the AIM 65 board all operate with ASCII codes. When you press a key on the keyboard, an 8-bit ASCII code is placed on the system's data bus; similarly, the printer and display assume that the information on the data bus is the ASCII code that represents the character to be printed or displayed.

The R6502 microprocessor operates on numeric data in either of two forms, binary or BCD (Binary-Coded Decimal), depending on the state of the Decimal Mode (D) bit in the Processor Status Register. It does not automatically convert this data into ASCII (or any other) form for output to a peripheral device, nor does it automatically convert ASCII character from an input device into binary or BCD. How, then, is the information converted from one form to another? Many computer systems include special electronic circuits that perform the conversions in hardware, but most general-purpose systems (like the AIM 65) perform the conversions in software, with specially-designed conversion subroutines. In this experiment, we will discuss subroutines that accept ASCII-based characters from the keyboard and convert them into binary data in memory. In Experiment 12 we will convert binary data in memory to one or more ASCII characters, and output these characters to the AIM 65's display and printer.

Figure 11-1 summarizes the ASCII character set, showing the most-significant digit (MSD) and the least-significant digit (LSD) of the hexadecimal representation for each character. For example, the character B has a hexadecimal representation of 42 (MSD = 4, LSD = 2).

	MSD	0	1	2	3	4	5	6	7
LSD		000	001	010	011	100	101	110	111
0	0000	NUL	DLE	SP	0	@	P		p
1	0001	SOH	DC1	!	1	A	Q	a	q
2	0010	STX	DC2	"	2	B	R	b	r
3	0011	ETX	DC3	#	3	C	S	c	s
4	0100	EOT	DC4	$	4	D	T	d	t
5	0101	ENQ	NAK	%	5	E	U	e	u
6	0110	ACK	SYN	&	6	F	V	f	v
7	0111	BEL	ETB	'	7	G	W	g	w
8	1000	BS	CAN	(	8	H	X	h	x
9	1001	HT	EM	)	9	I	Y	i	y
A	1010	LF	SUB	*	:	J	Z	j	z
B	1011	VT	ESC	+	;	K	[	k	{
C	1100	FF	FS	,	<	L	\	l	|
D	1101	CR	GS	–	=	M	]	m	}
E	1110	SO	RS	•	>	N	↑	n	~
F	1111	SI	VS	/	?	O	←	o	DEL

Figure 11-1. ASCII Character Set
(Courtesy of Rockwell International)

Input Subroutine

Since the primary task of Experiments 11 and 12 is to study ways to convert data from ASCII to binary, and vice versa, we will not concern ourselves with how the data is transferred between the R6502 and external input and output devices. Instead, we will forego that discussion until Experiment 13, and use subroutines in the AIM 65 Monitor to perform the data transfer operations.

To input a keyboard character, your program should call the AIM 65's REDOUT subroutine. This subroutine waits for a key to be pressed, and then inputs the ASCII character corresponding to that key into the accumulator, and both prints it and displays it. The REDOUT subroutine begins at location $E973, so your program can call it with the instruction

```
JSR E973
```

Types of Input Conversions

If we confine our discussion to converting ASCII-based input data (as we will here) to the two forms that the R6502 can process -- binary and BCD -- and observe that data can be entered in either hexadecimal or decimal form, we will observe that there are essentially three types of conversions to be considered. They are:

- ASCII-based hexadecimal to binary
- ASCII-based decimal to binary
- ASCII-based decimal to BCD

(The fourth combination, ASCII-based hexadecimal to BCD, is so rare that we will disregard it.)

ASCII-Based Hexadecimal to Binary

Since a substantial amount of R6502 programming is done using hexadecimal data, many applications require hexadecimal data being entered from the keyboard to be converted to binary. Table 11-1 shows the binary and ASCII form for each of the 16 hexadecimal characters.

Table 11-1 shows us what conversion is needed, and gives us a few ideas about how to do it. For starters, note that for the numeric values, 0 to 9, the low-order hex digit of the ASCII value is identical to the hex character. Obviously, all that must be done to convert this range of values is to mask out (strip off) the high-order hex digit of the ASCII value, which is a 3. For the remaining values, A to F, if we mask out the ASCII value's high-order digit, 4, the low-order digit is exactly nine less than the required hex value ($1 + $9 = $A, $2 + $9 = $B, and so on).

Most ASCII-based-hex-to-binary conversion subroutines convert two ASCII-based hexadecimal characters to an eight-bit binary number. Why are two characters converted? Because each hexadecimal character converts to a four-bit binary value (0000 through 1111); so rather than converting one 8-bit ASCII character to one 4-bit binary value in memory (and wasting four bits in each memory location), we convert two ASCII characters and "pack" the two resulting 4-bit binary values into a single memory location. The high-order digit value is stored in the four most-significant bits, the low-order digit value is stored in the four least-significant bits (Figure 11-2).

The AIM 65 Monitor contains an ASCII-based-hex-to-binary conversion subroutine. This subroutine, PACK (entry address $EA84), converts an ASCII character in the Accumulator to a 4-bit binary value in the four least-significant bits (LSBs), and clears the four most-significant bits (MSBs) to zero. If PACK is called a second time, this subroutine will move the first (high-order) digit to the four MSB, and place the new (least-significant) digit in the four LSBs.

Table 11-1. The Hexadecimal Numbering System

Hexadecimal Character	Binary Value	ASCII Value
0	0000	30
1	0001	31
2	0010	32
3	0011	33
4	0100	34
5	0101	35
6	0110	36
7	0111	37
8	1000	38
9	1001	39
A	1010	41
B	1011	42
C	1100	43
D	1101	44
E	1110	45
F	1111	46

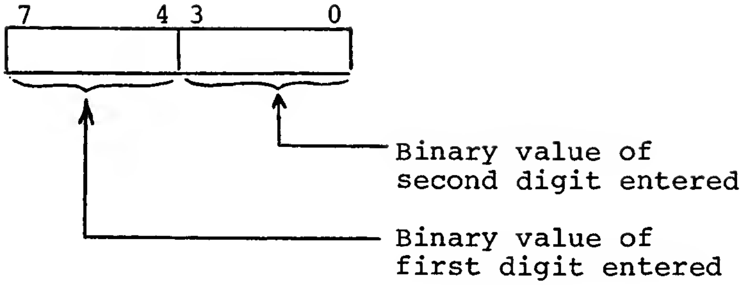

7
4 3
0
Binary value of second digit entered
Binary value of first digit entered

Figure 11-2. Two Converted ASCII Digits "Packed" into Eight-Bit Location

The PACK subroutine also checks each incoming ASCII character to determine whether or not it represents a hexadecimal value, and sets the Carry flag if a non-hex character has been entered.

ASCII-Based Decimal to Binary

In some respects, it is easier to convert ASCII-based decimal numbers to binary than to convert ASCII-based hexadecimal numbers to binary. However, as we shall see, the conversion process is more complex.

The ASCII decimal/binary relationships are included in Table 11-1, as the values 0 to 9. As you can see from this table, if we are working with decimal digits the only ASCII values that interest us are those that fall between $30 and $39; all values below $30 and above $39 do not represent decimal digits. You will also note that the binary equivalent of a decimal digit is nothing more than the four least-significant bits of the ASCII character.

An 8-bit register or memory location can hold the binary equivalent of decimal numbers between 0 and 255, so our basic ASCII-based-decimal-to-binary conversion subroutine must be able to accept up to three ASCII characters and convert them to an 8-bit binary value (as long as the digits do not form a number larger than 255).

As you already know, decimal numbers can be expressed as a series of integers multiplied by powers of 10. For example,

$$237 = (7 \times 1) + (3 \times 10) + (2 \times 100)$$

or

$$237 = (7 \times 10^0) + (3 \times 10^1) + (2 \times 10^2)$$

Since digits of a number are entered one at a time, there will have to be a multiply-by-10 routine (or subroutine) within a general-purpose ASCII-based-decimal-to-binary conversion subroutine. In this type of conversion routine, when a 93 is entered, the 9 must be multiplied by 10 before being added to the 3.

Of course, a number can be multiplied by 10 using a multiplication routine like you developed in Experiment 6. Alternatively, a number can be multiplied by 10 by taking advantage of the fact that left-shifting a number doubles the value of the number. Here is how that could be done: To multiply a number by 10,

- Left-shift the number twice (which multiplies it by four), then
- Add the original value to the result (which multiplies the number by five), then
- Left-shift that result (which multiplies the number by ten).

With this background, we can define the required sequence of operations for a three-digit, ASCII-based decimal to binary conversion subroutine:

- For the first digit, the conversion subroutine must input the digit, convert it to binary, and store the binary value as a partial result.
- For the second and third digits, the conversion subroutine must input the digit, convert it to binary, multiply the stored partial result by 10, and add the new digit to it (to update the partial result).

Since the AIM 65 accepts only hexadecimal data (unless you have an assembler), the AIM 65 Monitor has no ASCII-based decimal to binary conversion subroutine that you can call to perform that task. You will write such a conversion subroutine as an exercise in this experiment.

ASCII-Based Decimal to BCD

Since the decimal digits are a subset of the hexadecimal characters, the ASCII-decimal-to-binary conversion is similar to, but simpler than, the ASCII-hexadecimal-to-binary conversion described previously in this discussion. Although the AIM 65 Monitor has no ASCII-decimal-to-BCD conversion subroutine, its PACK subroutine, or some modified version of PACK, could be used to do the job.

PROCEDURE

1. You are to develop a program that inputs a decimal digit from the keyboard into the Accumulator, and converts it from ASCII to a four-bit binary value. Your program should input the keyboard digit by calling the AIM 65 Monitor's REDOUT subroutine (JSR E973), and should ignore any non-decimal character that is entered. Figure 11-3 shows the flowchart for this program.

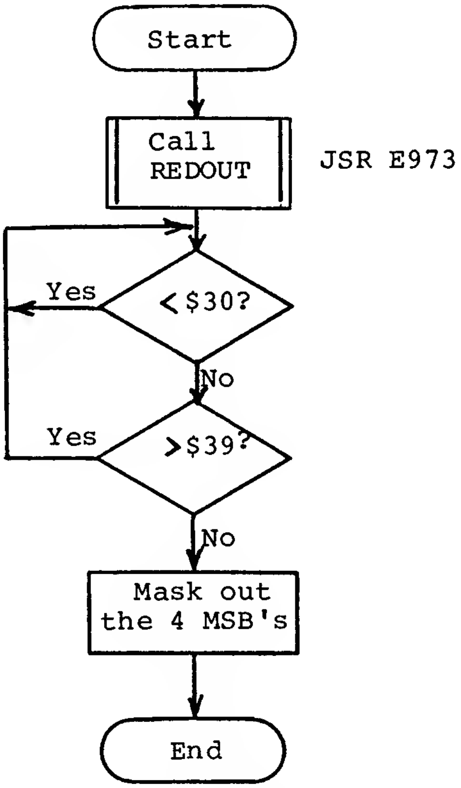

Start
Call REDOUT
JSR E973
< $30?
Yes
No
> $39?
Yes
No
Mask out the 4 MSB's
End

Figure 11-3. Keyboard Decimal Input Program

Write the code for this program below.

2. Run your program three times, pressing the key(s) indicated below, and record the Accumulator contents for each run.

Press Key(s)	Accumulator
9	
N, then 4	
2	

3. The program in Steps 1 and 2 processes only a single digit from the keyboard. You are now to develop a program that accepts three decimal digits from the keyboard, and converts them from ASCII to an eight-bit binary value in the Accumulator. Since the Accumulator is eight-bits wide, it can only represent decimal numbers between 000 and 255. If a number between 256 and 999 is entered, set the Carry flag before return to signify an error condition.

   For this new problem, leave your single-digit conversion program in memory, but make it into a subroutine (by adding an RTS). The new, three-digit conversion program will call this subroutine three times, once for each digit. This problem will also require another subroutine, one that multiplies the partial result by ten before it adds in the (converted) middle and low-order digits.

   Draw the flowchart below.

Write the program code below.

4. Run your program for the four cases listed below, and complete the table.

Entered Number	Result (Hex.)	Carry? (Yes/No)
000		
120		
255		
256		

12

CODE CONVERSION FOR OUTPUT

OBJECT

To learn the techniques for converting data to be output to external equipment.

DISCUSSION

In Experiment 11 we studied the types of operations a computer must perform to convert ASCII-based codes entered from the keyboard to the binary and BCD forms that the R6502 is able to process. In this experiment we will study the types of operations that a computer must perform to convert binary and BCD information to ASCII-based characters that are recognizable to many peripheral devices (including the AIM 65's display and printer). You should refer back to Figure 11-1 for a summary of the ASCII character set.

Output Subroutine

As in Experiment 11, we will use an AIM 65 Monitor subroutine to actually transfer the information between the R6502 and an external peripheral device.

To output an ASCII character to the display and printer, your program should call the AIM 65's OUTPUT subroutine. This character simply transmits the contents of the Accumulator to the display and printer -- it is up to you to ensure that the Accumulator contains the proper ASCII code when OUTPUT is called. The OUTPUT subroutine begins at location $E97A, so your program can call it with the instruction

```
JSR E97A
```

Types of Output Conversions

Because the R6502 can process two forms of numeric data -- binary and BCD -- and this data may be required for display in either its hexadecimal or decimal representation, we will observe that there are essentially three types of conversions to be considered. They are

- Binary to ASCII-based hexadecimal
- Binary to ASCII-based decimal
- BCD to ASCII-based decimal

(The fourth combination, BCD to ASCII-based hexadecimal, is so rare that we will disregard it.)

Binary to ASCII-Based Hexadecimal

As you will recall from Experiment 11, two consecutive ASCII-based hexadecimal characters were converted and "packed" into a single, 8-bit memory location. This packing was illustrated in Figure 11-2, which showed that the 4-bit representations of the most-significant (first entered) and least-significant (second entered) hexadecimal characters were packed into the high-order 4-bit positions and low-order 4-bit positions, respectively.

Two packed 4-bit values can, similarly, be converted to two ASCII-based hexadecimal characters by an "unpacking" sequence in which each 4-bit binary value is converted individually. The most common procedure is:

- Store the 8-bit value in memory (on the stack, if you like).
- Right-shift the Accumulator four times, to replace the most-significant digit's binary value in the four LSBs.
- Convert this value to ASCII-based hexadecimal. This is done by adding $30 if the value is between 0 and 9 and adding $37 if the value is between 10 and 15 (look at Table 11-1 if you are unsure why these values were used).
- Output the result, as the most-significant digit.
- Load the original value into the Accumulator.
- Mask out the four MSBs (AND $0F).
- Convert this second value to ASCII-based hexadecimal, by applying the rules given above.
- Output the result, as the least-significant digit.

Figure 12-1 shows a flowchart for a subroutine that performs the preceding sequence of operations. Since each of the BCD digits must be converted to ASCII and output, this subroutine calls another subroutine (also shown in Figure 12-1) which performs the conversion and outputs the result. Note that the second "call" to the conversion subroutine is not a call (JSR) at all, but a jump (JMP), which eliminates the need for an RTS instruction at the end of the main subroutine. Incidentally, the AIM 65 Monitor's NUMA subroutine (entry address $EA46) performs the same function as the subroutine in Figure 12-1, and outputs the two converted digits to the display and printer.

Binary to ASCII-Based Decimal

Although computers process numbers in their binary (or BCD) form, it is often desirable to print results in their decimal form, to help decimally-oriented human beings to understand them. Because this task is so common, it is worth discussing how binary numbers can be converted to their ASCII-based decimal equivalents.

Recall, from Experiment 11, that decimal numbers can be expressed as a series of integers multiplied by powers of 10. For example,

$$237 = (7 \times 10^0) + (3 \times 10^1) + (2 \times 10^2)$$

With this in mind, the primary job of converting an 8-bit binary number to a decimal number

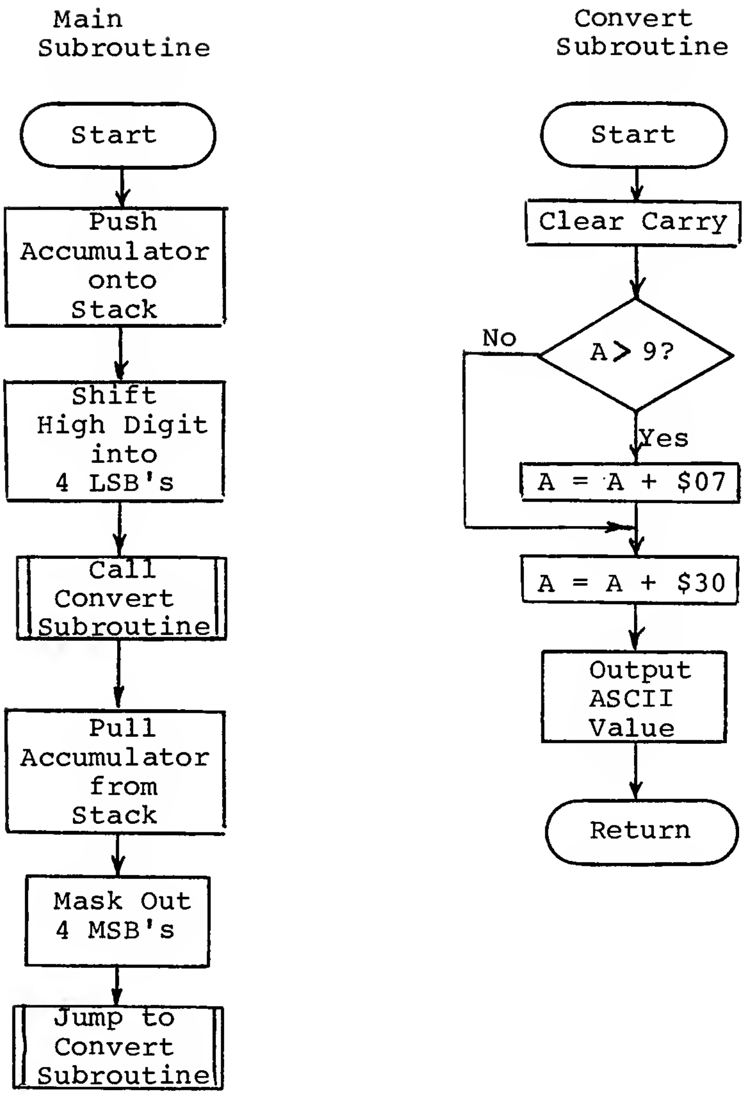

Main Subroutine
Start
Push Accumulator onto Stack
Shift High Digit into 4 LSB's
Call Convert Subroutine
Pull Accumulator from Stack
Mask Out 4 MSB's
Jump to Convert Subroutine
Convert Subroutine
Start
Clear Carry
A > 9?
No
Yes
A = A + $07
A = A + $30
Output ASCII Value
Return

Figure 12-1. Binary to ASCII-Based Hexadecimal Subroutine

reduces to finding out how many 100's, 10's and 1's are contained in the number. We can make this determination by performing a series of successive subtractions on the binary value. The number of times that 100 can be subtracted from the binary value tells us the 100's digit of the decimal representation. Similarly, the number of times that 10 can be subtracted from that remainder tells us the 10's digit of the decimal representation and the remainder of that operation tells us the 1's digit of the decimal representation.

For example, to determine the number of 100's in the decimal number 237 we will subtract 100 from 237 until another subtraction would produce a negative result. Each time 100 can be successfully subtracted, a 100's count should be incremented by 1. For the number 237, the sequence is:

$$\begin{array}{rl}
237 & \\
\underline{-100} & \\
137 & \text{100's count = 1} \\
\underline{-100} & \\
37 & \text{100's count = 2}
\end{array}$$

Now that the number of 100's is known, we can calculate a 10's count with a similar subtraction sequence. Working with our previous remainder, 37, the sequence is:

$$\begin{array}{rl}
37 & \\
\underline{-10} & \\
27 & \text{10's count = 1} \\
\underline{-10} & \\
17 & \text{10's count = 2} \\
\underline{-10} & \\
7 & \text{10's count = 3}
\end{array}$$

At this point, we need not perform any more subtractions since the remainder, 7, represents the number of units (the 1's count) in the answer.

The AIM 65 Monitor has no binary to ASCII-based decimal conversion subroutine. You will write such a subroutine as an exercise in this experiment.

BCD to ASCII-Based Decimal

Since the decimal digits (0 - 9) are included in the set of hexadecimal digits (0 - 9, A - F), converting BCD values to ASCII-based decimal is similar to--but simpler than--converting binary values to ASCII-based hexadecimal. To convert BCD values, you would use the general subroutine shown in Figure 12-1, but its "convert" subroutine would always add $30 to the Accumulator, with no need to check whether the value in the Accumulator (A) is greater than 9.

PROCEDURE

1. Write a program that converts an 8-bit binary value in the Accumulator to a three-digit, ASCII-based decimal string that is output to the AIM 65's display and printer. Your subroutine should use the AIM 65 Monitor's OUTPUT subroutine (JSR E97A) to perform the display operation.

   Draw the flowchart below.

   Hint: Use a compare instruction to determine whether the count value (100 or 10) can be subtracted from the remainder. If it can, perform the subtraction; if not, output the count and continue processing with the next decimal "weight."

Write the program code below.

2. Run your program for the four cases listed below, and complete the table.

Number in Accumulator		Printed Result
Dec.	Hex.	
030		
255		
002		
126		

13

INPUT/OUTPUT

OBJECT

To learn the philosophy of the 6500 system's "memory-mapped" I/O structure, and how to apply it in controlling a simple output device, the AIM 65 display.

PRE-LAB PREPARATION

Read Section 8.1 of the AIM 65 User's Guide, to gain a basic understanding of input/output programming, then read Section 3.1 of the R6500 Hardware Manual to learn about the 6500 system I/O structure.

In this experiment, you will be putting information on the AIM 65 display. The display communicates with the R6502 CPU through an R6520 Peripheral Interface Adapter (PIA) integrated circuit, so you should also read Section 5 of the R6500 Hardware Manual, Section 11.0 of the R6500 Programming Manual and Section 7.2.7 of the AIM 65 User's Guide for information on the R6520.

DISCUSSION

R6502 Input/Output Structure

The R6502 is an 8-bit microprocessor, which means that the basic unit of information, the byte, is eight bits wide. Further, all information is transferred to and from memory and input/output devices eight bits at a time. To transfer more than eight bits requires additional transfer operations. All information transfers between the R6502 and external devices are conducted on the 8-bit Data Bus. The Data Bus is bidirectional, so the same lines are used to transfer information into and out of the R6502.

Since a microcomputer system may include a variety of memory and I/O circuits, how does the R6502 differentiate whether it wants to communicate with memory or with an I/O circuit? The answer is that the R6502, unlike many other microprocessors, makes no differentiation between memory and I/O devices -- it treats every external device as memory! For this reason, the R6502 is said to have a "memory-mapped" I/O structure.

Because of its memory-mapped I/O, the R6502 has no special input or output instructions in its instruction set. In fact, the same Load instructions that read information from memory are used to input information from a peripheral device. Similarly, the same Store instructions that write information into memory are used to output information to a peripheral device. For this reason, each peripheral device has one or more unique addresses in the system. For example, the R6520 PIA that interfaces to the AIM 65 display has four unique addresses--$AC00 through $AC03--that you will be using to communicate with the display.

The R6502 transmits addresses to memory and peripheral devices on 16 lines that are collectively known as the Address Bus. Being 16 bits wide, the Address Bus can select any of up to 65,536 (64K) locations. All external devices, both memory circuits and I/O circuits, must be connected to the Address Bus.

The R6520 Peripheral Interface Adapter (PIA)

Perhaps the easiest way to learn about programming input/output operations for a microcomputer system is to actually apply the principles you have read about to a "real" peripheral device. In this experiment, you will apply those principles to a fairly simple peripheral device that is mounted on the AIM 65 circuit board -- the display.

As you learned in Section 7.2.7 of the AIM 65 User's Guide, the AIM 65's display is connected, or interfaced, to the system through a widely used, general-purpose integrated circuit, the R6520 PIA. The PIA, shown in Figure 13-1, has two 8-bit bidirectional I/O ports (Port A and Port B) and four peripheral control/interrupt lines (CA1, CA2, CB1 and CB2), two for each port. Each of the 16 lines that form the two I/O ports can be selected, under program control, to function as either an input line or an output line.

Six of the PIA's internal registers are addressable: Peripheral Interface Buffers A and B, Data Direction Registers DDRA and DDRB, and Control Registers CRA and CRB. The register addressing is performed by two Register Select lines, RS0 and RS1, which are normally connected to low-order address lines A0 and A1, respectively. As you know, two address lines allow addressing of only four different locations in memory. Two of these locations are dedicated to addressing the Control Registers. Each of the other two locations are shared by a Peripheral Interface Buffer and a Data Direction Register. Bit 2 of each Control Register determines whether the associated port's Peripheral Address Buffer (Bit 2 = 1) or Data Direction Register (Bit 2 = 0) is addressed by Register Select Lines RS0 and RS1.

This address sharing imposes no inconvenience on the programmer, because the Data Direction Registers must be accessible only at initialization time, when they are used to assign individual port lines as inputs or outputs. Once the Data Direction Registers have been initialized, their contents remain constant and the shared addresses are used to write data into, or read data out of, the Peripheral Interface Buffers.

Use of the R6520 PIA With the AIM 65 Display

From your reading you know that the R6520 PIA is a general-purpose interface device that can be programmed to communicate with a variety of different kinds of peripheral I/O equipment. It is capable of passing status and control signals between the R6502 CPU and the attached peripheral units (such signals as "data accepted" and "ready to receive data") and having its individual data pins configured in any combination of inputs and outputs. However, the AIM 65 display is so simple that very few of these sophisticated options are utilized. In fact, with the display, the PIA is used as a simple, output-only interface device, and does little more than pass character codes, display select signals, digit addresses and a "write" control signal to the display.

In the AIM 65 display interface, the R6520 PIA responds to these four addresses:

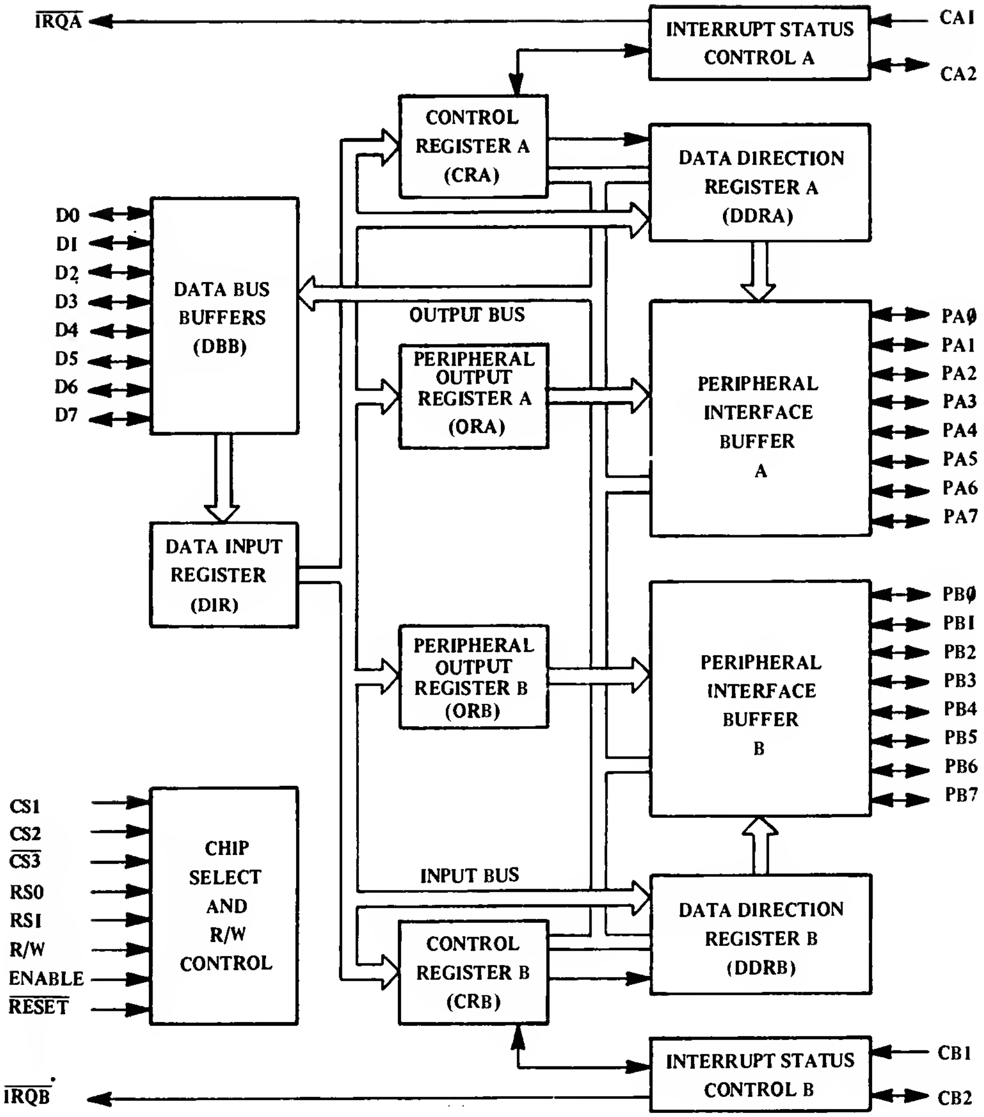

IRQA
INTERRUPT STATUS CONTROL A
CA1
CA2
CONTROL REGISTER A (CRA)
DATA DIRECTION REGISTER A (DDRA)
D0
D1
D2
D3
D4
D5
D6
D7
DATA BUS BUFFERS (DBB)
OUTPUT BUS
PERIPHERAL OUTPUT REGISTER A (ORA)
PERIPHERAL INTERFACE BUFFER A
PA0
PA1
PA2
PA3
PA4
PA5
PA6
PA7
DATA INPUT REGISTER (DIR)
PERIPHERAL OUTPUT REGISTER B (ORB)
PERIPHERAL INTERFACE BUFFER B
PB0
PB1
PB2
PB3
PB4
PB5
PB6
PB7
CS1
CS2
CS3
RS0
RS1
R/W
ENABLE
RESET
CHIP SELECT AND R/W CONTROL
INPUT BUS
DATA DIRECTION REGISTER B (DDRB)
CONTROL REGISTER B (CRB)
INTERRUPT STATUS CONTROL B
CB1
CB2
IRQB

Figure 13-1. Block Diagram of the R6520 PIA
(Courtesy of Rockwell International)

- $AC00 is used to access Peripheral Interface Buffer A (called Data Register A in the AIM 65 User's Guide) and Data Direction Register A
- $AC01 is used to access Control Register A
- $AC02 is used to access Peripheral Interface Buffer B (called Data Register B in the AIM 65 User's Guide) and Data Direction Register B
- $AC03 is used to access Control Register B

Actually, you will only have to concern yourself with two of the addresses, $AC00 and $AC02 (the shared addresses), in this experiment because the AIM 65 Monitor program initializes the PIA's Data Direction Registers and Control Registers automatically, when you press the RESET button. Students who wish to follow this PIA initialization procedure will find it on Lines 0394 through 0408 in the AIM 65 Monitor Program Listing, but basically here is what it does:

- It clears both Control Registers to zero, which disables interrupts and handshaking, and selects the Data Direction Registers (Bit 2 = 0). Then,
- It loads $FF into both Data Direction Registers, which assigns both I/O ports as output-only ports. Then,
- It loads $04 into both Control Registers, which keeps interrupts and handshaking disabled, but selects the Peripheral Interface Buffers for subsequent addressing (Bit 2 = 1).

With this initialization completed, neither the Data Direction Registers nor the Control Registers will require alteration and, as just mentioned, all of your display programming will be confined to addressing Peripheral Interface Buffers A and B--addresses $AC00 and $AC02, respectively.

Programming the R6520 PIA for Display

In the AIM 65 display interface, the PIA's Peripheral Interface Buffer A is used to hold the display address and Peripheral Interface Buffer B is used to hold the character data. The format of both registers is summarized in Figure 13-2.

Peripheral Interface Buffer A is divided into three fields: Digit Address (Bits 0 and 1), Display $\overline{\text{Select}}$ (Bits 2 through 6) and Display $\overline{\text{Write}}$ (Bit 7). The bars over the words "Select" and "Write" indicate that the function is activated by a 0 in a given bit position. The AIM 65's 20-character display is comprised of five, 4-digit display units. The Digit Address and Display $\overline{\text{Select}}$ fields select the digit position and display unit, respectively, on which the character is to be displayed. As you can see in Figure 13-2, there are only five "valid" binary combinations for the Display $\overline{\text{Select}}$ field, one combination for each of the five display units. Each digit position of the AIM 65's five display units has a register than can hold character data for that digit position. This register is loaded with the contents of Peripheral Interface Buffer B when your program executes a Store instruction with Display $\overline{\text{Write}}$ reset to 0 (Bit 7 = 0), and holds that character data for continuous display with Display $\overline{\text{Write}}$ set to 1 (Bit 7 = 1).

Observe from Figure 13-2 that the low-order seven bits of Peripheral Interface Buffer A (the Digit Address and Display $\overline{\text{Select}}$) constitute a Display Position Address. Table 13-1 shows the Display Position Addresses for the 20 digit positions in the display.

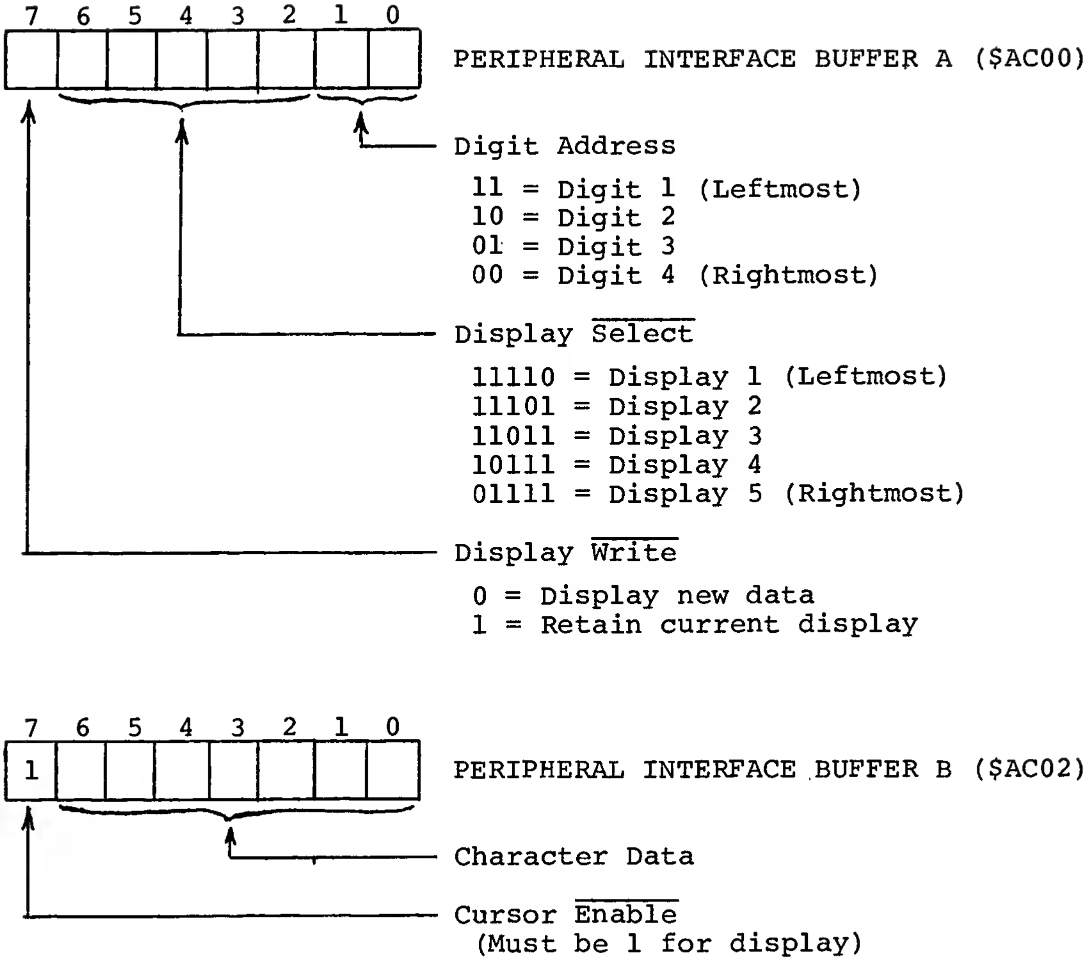

7 6 5 4 3 2 1 0
PERIPHERAL INTERFACE BUFFER A ($AC00)
Digit Address
11 = Digit 1 (Leftmost)
10 = Digit 2
01 = Digit 3
00 = Digit 4 (Rightmost)
Display $\overline{\text{Select}}$
11110 = Display 1 (Leftmost)
11101 = Display 2
11011 = Display 3
10111 = Display 4
01111 = Display 5 (Rightmost)
Display $\overline{\text{Write}}$
0 = Display new data
1 = Retain current display
7 6 5 4 3 2 1 0
1
PERIPHERAL INTERFACE BUFFER B ($AC02)
Character Data
Cursor $\overline{\text{Enable}}$
(Must be 1 for display)

Figure 13-2. Peripheral Interface Buffers for Display PIA

Table 13-1. AIM 65 Display Position Addresses

DISPLAY UNIT	DIGIT NUMBER			
	1	2	3	4
1	7B	7A	79	78
2	77	76	75	74
3	6F	6E	6D	6C
4	5F	5E	5D	5C
5	3F	3E	3D	3C

Peripheral Interface Buffer B holds the ASCII bit pattern for the character to be displayed. The AIM 65 can display 64 different characters--exactly one-half of the 128-character ASCII set that we encountered in Figure 11-1 of Experiment 11--including numeric characters 0 through 9, the upper-case alphabetic characters and 28 symbol characters (including a blank space). The codes for the valid display characters are summarized in Table 13-2. Table 7-6 in the AIM 65 User's Guide shows how these characters actually appear on the display.

Here, then, is the procedure to display information:

1. Load a byte of character data into a register, and write it into Peripheral Interface Buffer B (location $AC02).

2. Load Display Position Address into a register, and write it into Peripheral Interface Buffer A (location $AC00). Bit 7 must be a 0, to display the new character.

3. Load the value $FF into a register, and write it into Peripheral Interface Buffer A ($AC00). This "locks" the character into the digit's holding register and deselects that particular digit position.

4. If another character is to be displayed, return to Step 1.

Table 13-2. AIM Display Character Set

	MSD	A	B	C	D
LSD		1010	1011	1100	1101
0	0000	(Sp)	0	@	P
1	0001	!	1	A	Q
2	0010	"	2	B	R
3	0011	#	3	C	S
4	0100	$	4	D	T
5	0101	%	5	E	U
6	0110	&	6	F	V
7	0111	'	7	G	W
8	1000	(	8	H	X
9	1001	)	9	I	Y
A	1010	*	:	J	Z
B	1011	+	;	K	[
C	1100	,	<	L	\
D	1101	-	=	M	]
E	1110	.	>	N	^
F	1111	/	?	O	_

PROCEDURE

1. Write a program that displays the letter M in the left-most position of the AIM 65 Display. End your program with a Jump (JMP) instruction that jumps to itself, producing an infinite loop, and keeping the display intact until you press the RESET push-button. Run the program to verify that it works.

2. Now, write a program that will display your name on the AIM 65. The program will access two data tables that you must store in memory:

   - The ASCII display characters (Table 13-2) that form your name should be stored in a table starting at location \$0300. The last entry in the table should be an ASCII Carriage Return character (\$0D), which the program will use as an end-of-text indicator.

   - The AIM 65 Display Position Addresses (Table 13-1) should be stored in another table, starting at location \$0400.

   Flowchart the problem on the next page. You will find the four-step procedure at the end of the Discussion helpful in making up this flowchart.

Flowchart for name display program.

Write the program code for the name display program below, then enter it into memory, starting at location $0200. Upon encountering the $0D end-of-text indicator, the program should wait in an infinite loop until you press the Reset push-button--as you did in Step 1.

3. List the contents of the text buffer in the space below, then enter that information into memory, starting at location $0300. Note: Restrict the text to 20 display positions (abbreviate your first name, if necessary) and put a blank space between your first name and surname.

Position	Alphabetic Character	ASCII Character (hex)
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		
12		
13		
14		
15		
16		
17		
18		
19		
20		

4. Run your program, to verify that it works.

14

A MORE POWERFUL I/O DEVICE, THE R6522 VIA

OBJECT

To learn about the counting and timing features of a more complex, more flexible, I/O device, the R6522 VIA.

PRE-LAB PREPARATION

Read Sections 8.1, 8.2, 8.3, 8.6 and 8.7 of the AIM 65 User's Guide.

DISCUSSION

The AIM 65 microcomputer includes an input/output device that is exclusively available to the user. It is initialized by the AIM 65 Monitor at power-up time, but is otherwise unused by the AIM 65 system software. This device, the R6522 Versatile Interface Adapter (VIA) (Figure 14-1), is an enhanced version of the R6520 PIA device, which we studied in Experiment 13. The R6522 VIA provides all of the features of the R6520 PIA, but also has a pair of programmable timer/counters, a shift register and a more highly sophisticated "handshaking" capability.

This experiment is intended to highlight the R6522 VIA's timing and counting capabilities, which are provided by its two built-in timer/counters, called Timer 1 and Timer 2. From the pre-lab reading you can appreciate that these timers have potential in various applications that involve either counting external pulses or generating time intervals (to produce time delays or, perhaps, a time-of-day clock). An especially attractive feature of both Timer 1 and Timer 2 is that once they are configured, they run unattended--totally independent of the R6502 CPU!

By way of introducing the timing capabilities of the R6522 VIA, this experiment requires you to develop a program that measures the elapsed time between two events: Pressing the G key, to start the timing program, and pressing the Reset push-button, to stop the program. The elapsed time must be accurate to 1/100th of a second; that is, it must be accurate to 10 milliseconds.

R6522 Features Used to Program the Problem

Very simply, the preceding problem requires us to have a program that will start when the G key is pressed, and end when the Reset button is pressed, and measure the elapsed time (in 10-millisecond increments) between these two events. Since there is no defined

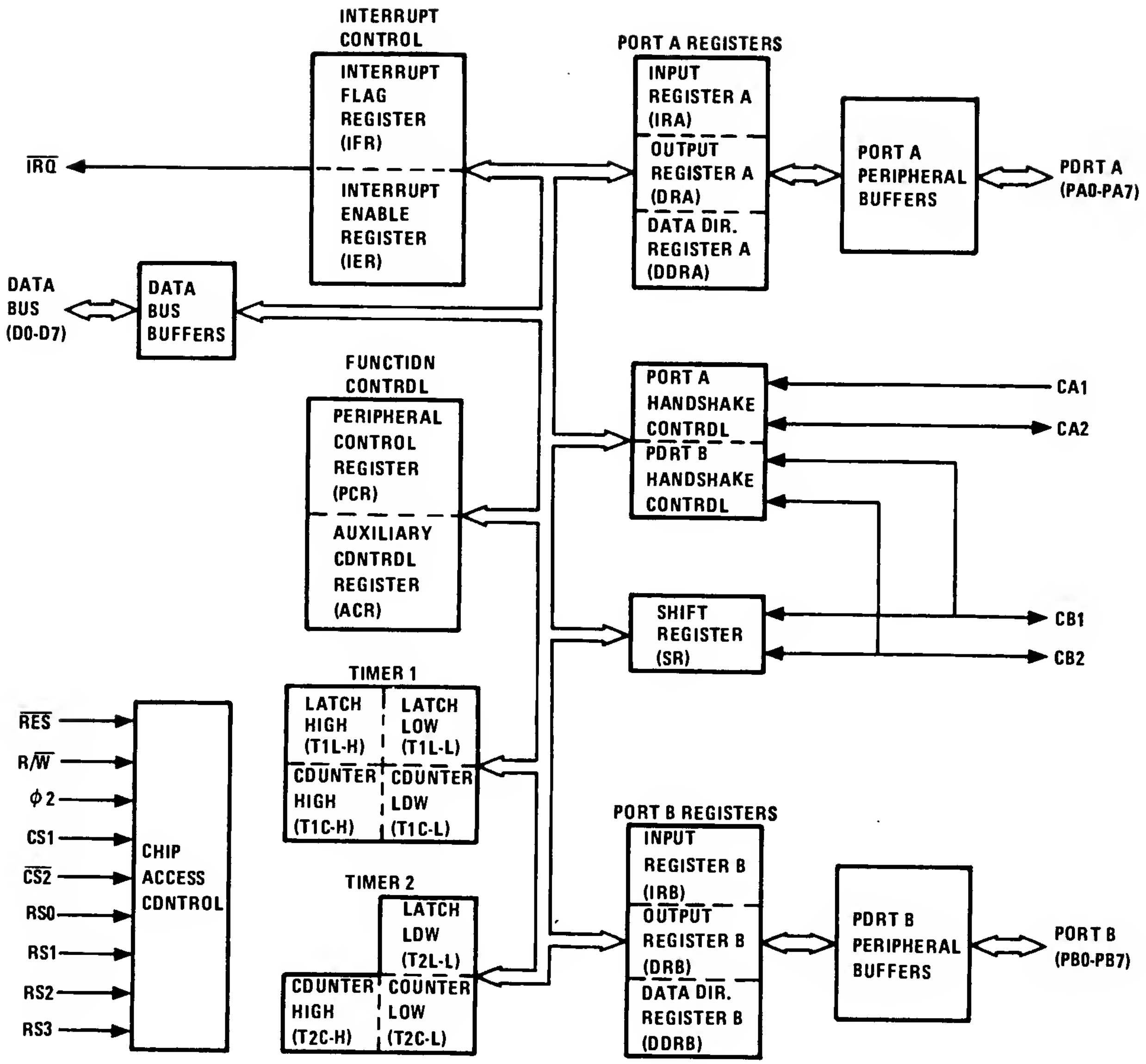

INTERRUPT CONTROL
INTERRUPT FLAG REGISTER (IFR)
INTERRUPT ENABLE REGISTER (IER)
$\overline{IRQ}$
DATA BUS (D0-D7)
DATA BUS BUFFERS
FUNCTION CONTROL
PERIPHERAL CONTROL REGISTER (PCR)
AUXILIARY CONTROL REGISTER (ACR)
TIMER 1
LATCH HIGH (T1L-H)
LATCH LOW (T1L-L)
COUNTER HIGH (T1C-H)
COUNTER LOW (T1C-L)
TIMER 2
LATCH LOW (T2L-L)
COUNTER HIGH (T2C-H)
COUNTER LOW (T2C-L)
$\overline{RES}$
R/$\overline{W}$
ϕ2
CS1
$\overline{CS2}$
RS0
RS1
RS2
RS3
CHIP ACCESS CONTROL
PORT A REGISTERS
INPUT REGISTER A (IRA)
OUTPUT REGISTER A (ORA)
DATA DIR. REGISTER A (DDRA)
PORT A PERIPHERAL BUFFERS
PORT A (PA0-PA7)
PORT A HANDSHAKE CONTROL
PORT B HANDSHAKE CONTROL
CA1
CA2
SHIFT REGISTER (SR)
CB1
CB2
PORT B REGISTERS
INPUT REGISTER B (IRB)
OUTPUT REGISTER B (ORB)
DATA DIR. REGISTER B (DDRB)
PORT B PERIPHERAL BUFFERS
PORT B (PB0-PB7)

Figure 14-1. Block Diagram of the R6522 VIA
(Courtesy of Rockwell International)

way of knowing when the Reset button will be pressed, the program must be capable of monitoring the elapsed time continuously, at 10-millisecond intervals, until someone presses Reset.

As you know from reading Section 8.7 of the AIM 65 User's Guide, Timers 1 and 2 both have "one-shot" timing modes, which will produce a specified time interval one time. These modes could certainly be employed to solve our problem, but that would be inefficient, since the timer would have to be re-started after each 10-millisecond interval. A much more desirable candidate for the solution is the "free-running" mode of Timer 1, which will automatically re-initialize after each time interval, thereby producing continuous time intervals until it is halted. Therefore, for this experiment we need to know:

1. How to put Timer 1 in the free-running mode
2. How to load a count value (10 milliseconds) into Timer 1
3. How to start Timer 1
4. How to detect whether or not Timer 1 has timed out

Accessing the VIA Registers

As you know, the user-dedicated R6522 VIA responds to 16 memory addresses (that is, it "occupies" 16 locations in Memory)--$A000 through $A00F--as summarized in Table 8-1 of the AIM 65 User's Guide. To perform the four operations just listed for our lab problem, we will need to access the locations that contain Timer 1's Auxiliary Control Register (to select the operating mode), its latches (high and low) and counters (high and low), and its Interrupt Flag Register (to detect the timeout). Table 14-1 is a summary of the addressing for the R6522 VIA locations that are applicable to this experiment. (Two other locations, $A006 and $A007, also apply to Timer 1, but they are used to change the count value while Timer 1 is running, an option that is not needed in this experiment.)

Table 14-1. R6522 Memory Assignments for Experiment 14

Location	Register Selected	
	Write Operations	Read Operations
$A004	Write Timer 1 Latch low byte	Read Timer 1 Counter low byte Clear T1 Interrupt Flag (IFR6)
$A005	Write Timer 1 Latch high byte Clear T1 Interrupt Flag (IFR6) Initiate counting	Read Timer 1 Counter high byte
$A00B	Write Auxiliary Control Reg.	Read Auxiliary Control Reg.
$A00D	Write Interrupt Flag Reg.	Read Interrupt Flag Reg.

Program Instructions For VIA Timer 1

The VIA's timer modes are selected based on the control information that you enter into the Auxiliary Control Register, location $A00B. As you can see from Figure 14-2, the Control Field for Timer 1, the two high-order bits of the Auxiliary Control Register, permits you to specify whether Timer 1 is to operate in its one-shot or free-running mode, and whether or not a timeout will cause an external signal is to be output on the VIA's Pin PB7. For our lab timing problem, decide what value should be written into the Auxiliary Control Register, and note the value on page 112.

Auxiliary Control Register ($A00B) =

Why did you choose this particular value? Answer below.

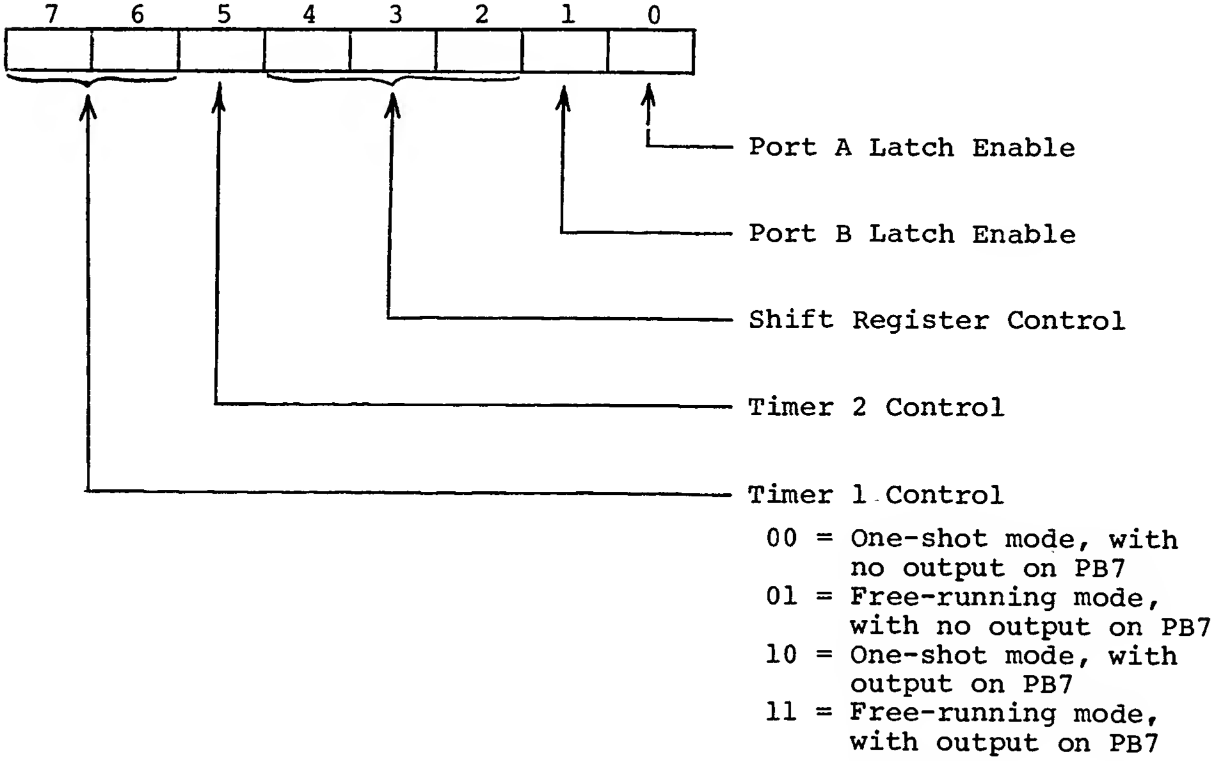

7
6
5
4
3
2
1
0
Port A Latch Enable
Port B Latch Enable
Shift Register Control
Timer 2 Control
Timer 1 Control
00 = One-shot mode, with no output on PB7
01 = Free-running mode, with no output on PB7
10 = One-shot mode, with output on PB7
11 = Free-running mode, with output on PB7

Figure 14-2. VIA Auxiliary Control Register ($A00B)

What count value must be written into Timer 1's Latch locations, $A004 and $A005, to produce a 10-millisecond interval? Timer 1 decrements once every microsecond (1/1000 milliseconds), and has a two-cycle "overhead," during which the Latch values are loaded into the Counters. Therefore, Timer 1's Latches should be initialized with the desired count value (in microseconds), minus two. In this experiment's lab problem, we want to calculate a time-until-Reset that is accurate to 1/100 of a second (0.01 seconds), so we should use a count value that produces a 10-millisecond interval. Record the count values on page 113.

For a 10-millisecond time interval:

Timer 1 Latch, low ($A004) =

Timer 1 Latch, high ($A005) =

In terms of the R6522 all that remains to be defined is how to determine when Timer 1 has timed out (i.e. 10 milliseconds have elapsed). Timeout is detected by checking the status of the VIA's Interrupt Flag Register (location $A00D). The Interrupt Flag Register's seven low-order bit positions, Bits 0 through 6, hold the status of seven different VIA conditions, as shown in Figure 14-3. (These bits can also be used to generate an "interrupt request" to the R6502 CPU; interrupts will be discussed in Experiment 15.)

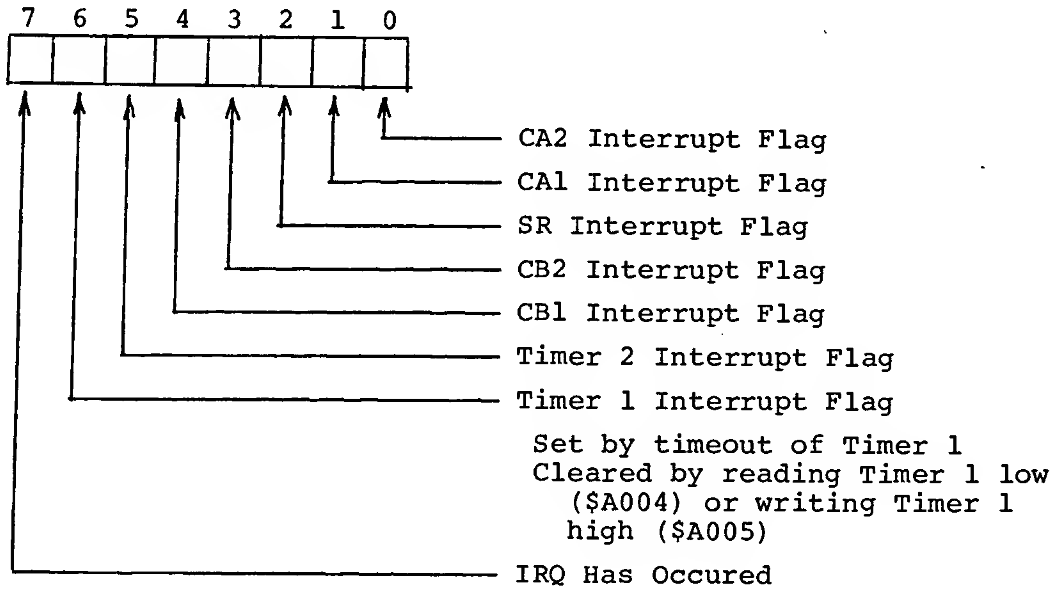

7 6 5 4 3 2 1 0
CA2 Interrupt Flag
CA1 Interrupt Flag
SR Interrupt Flag
CB2 Interrupt Flag
CB1 Interrupt Flag
Timer 2 Interrupt Flag
Timer 1 Interrupt Flag
Set by timeout of Timer 1
Cleared by reading Timer 1 low ($A004) or writing Timer 1 high ($A005)
IRQ Has Occured

Figure 14-3. VIA Interrupt Flag Register ($A00D)

You will note that Bit 6 of the Interrupt Flag Register holds the status of Timer 1. This bit will be set to 1 when Timer 1 times out (10 milliseconds elapse, in our application), and will stay set until the program either reads the contents of the Timer Counter low byte (location $A004) or writes a new count value into the Timer 1 Latch high byte (location $A005). Therefore, your program must perform two operations to properly service Timer 1:

- It must check for timeout, by looking for a 1 on IFR6.
- When timeout occurs (IFR6 = 1), it must read the contents of location $A004, to clear the interrupt flag.

Armed with the preceding background information and your pre-lab reading, you should now know how to perform the four Timer 1 programming tasks that were defined earlier in this discussion. The solutions are:

1. How to put Timer 1 in the free-running mode. Solution: The free-running mode is selected through the Auxiliary Control Register (location $A00B).
2. How to load a count value into Timer 1. Solution: The count value is entered by storing the count minus two, in microseconds, into the low and high bytes of the Timer 1 Latches (locations $A004 and $A005).
3. How to start Timer 1. Solution: Timer 1 automatically starts when the high-order count value is stored.
4. How to detect whether or not Timer 1 has timed out. Solution: Timer 1 timeout is indicated when Bit 6 of the Interrupt Flag Register (location $A00D) is 1.

PROCEDURE

1. As outlined in the Discussion, you are to write a program that measures elapsed time until you press the Reset push-button on the AIM 65. The time must be accurate to 1/100 of a second (10 milliseconds). The timing is performed by an R6522 VIA timer/counter, called Timer 1. The time count, recorded in memory, is incremented each time Timer 1 times out.

   Figure 14-4 shows a flowchart for the timing problem. Write the code for the program in the space below. Use memory locations $20 and $21 to hold the low and high byte of the time count, respectively.

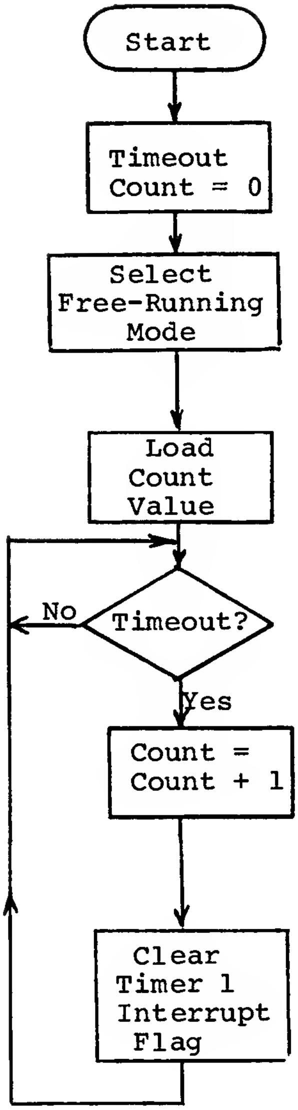

Start
Timeout Count = 0
Select Free-Running Mode
Load Count Value
Timeout?
No
Yes
Count = Count + 1
Clear Timer 1 Interrupt Flag

Figure 14-4. Flowchart for Timing Program

2. If your program is working correctly, a ten-second interval should produce a value of decimal 1,000 ($03E8) in the count locations, $20 and $21. Put your AIM 65 in the Run mode, then execute the program and press Reset after ten seconds have elapsed; use a wrist watch or sweep second hand of a wall clock as the time standard. Make three runs of your program, pressing Reset after ten seconds each time, and record your results below.

"Correct" Count Value		Observed Count Values					
		Pass #1		Pass #2		Pass #3	
$20	$21	$20	$21	$20	$21	$20	$21
E8	03						

3. What is the maximum time, in seconds, that can be properly recorded in locations $20 and $21? Enter your answer below.

Maximum total time (seconds) =

15

INTERRUPTS

OBJECT

To learn about the R6502's interrupt structure, and see how an interrupt can be generated by an external device, the AIM 65's user-dedicated R6522 VIA.

PRE-LAB PREPARATION

Read Chapter 9 of the Programming Manual, then read Section 7.8 of the AIM 65 User's Guide.

DISCUSSION

In the last experiment we developed a program that used the AIM 65's user-dedicated R6522 VIA to count at 0.01-second intervals until Reset is pressed. Although the VIA's Timer 1 ran independent of the R6502, the program in Experiment 14 required the R6502 to continuously check, or poll, the Interrupt Flag Register to find out whether or not the 0.01-second interval had elapsed.

Polling a device is one way to find out if a certain event has occurred, but it has the disadvantage of requiring the R6502 to spend valuable processing time just wating for the occurrence. In fact, this kind of polling is analagous to sitting by the telephone and picking up the receiver from time to time to find out whether anyone is calling you. Just as a telephone has a bell to notify you when you are being called, interface devices (like the R6522 VIA) have a way to notify the R6502 that they are "calling" it. The way interface devices call the R6502 is with an interrupt.

Types of Interrupts

What is an interrupt? An interrupt is an externally-generated signal that temporarily suspends the program being executed by the microprocessor, and transfers control to a subroutine that is designed to service that particular interrupt. Some devices use interrupts to inform the microprocessor that a particular event has occurred (e.g., a timer has elapsed), other devices use interrupts to inform the microprocessor that they have data to be input (e.g., a tape drive) or need data from the microprocessor (e.g., a graphic plotter).

Interrupts are also used to notify the R6502 of some condition that requires immediate attention, such as a power failure.

You'll note, then that we have just described two different types of interrupts: those that can be temporarily ignored by the microprocessor and those that need its immediate attention. If the first kind of interrupt can be compared to a ringing telephone (which you can ignore, if you wish), the second kind can be compared to a car's tire blowing out (which you must deal with immediately). The R6502 can accomodate either type of interrupt, and differentiates them based on which of two input lines is activated. Peripheral devices that wish to notify the R6502 of an event that can be temporarily ignored do so by activating the $\overline{\text{IRQ}}$ (Interrupt Request) line. Peripheral devices that wish to notify the R6502 of an event that must be serviced immediately do so by activating a different line, $\overline{\text{NMI}}$ (Non-Maskable Interrupt).

The R6502 will process a non-maskable interrupt (on $\overline{\text{NMI}}$) as soon as it has finished executing the current instruction. However, the R6502 may or may not process an interrupt request (on $\overline{\text{IRQ}}$) immediately, depending on the state of the Interrupt Disable (I) bit in the Processor Status Register. If I is reset to 0 when an $\overline{\text{IRQ}}$ is received, the R6502 will process the interrupt request; if I is set to 1 when an $\overline{\text{IRQ}}$ is received, the R6502 will ignore the interrupt request until I has been reset to 0. The R6502 instructions that control the state of the I bit are:

INSTRUCTION	DESCRIPTION
CLI	Clear Interrupt Disable Bit (to 0)
SEI	Set Interrupt Disable Bit (to 1)

How does the R6502 process $\overline{\text{IRQ}}$ and $\overline{\text{NMI}}$ interrupts? It does so by pushing the current contents of the Program Counter (high byte first, then low byte) and Processor Status Register onto the stack, then loading the Program Counter with the contents of two dedicated bytes in the microcomputer's Read-Only Memory (ROM). For an $\overline{\text{IRQ}}$ the Program Counter is loaded with the contents of locations $FFFE (low byte) and $FFFF (high byte). For an $\overline{\text{NMI}}$ the Program Counter is loaded with the contents of locations $FFFA (low byte) and $FFFB (high byte). These locations contain the addresses of interrupt service routines at which execution is to continue. In the AIM 65, locations $FFFE and $FFFF contain $78 and $E0, respectively, so $\overline{\text{IRQ}}$s cause the R6502 to fetch its next instruction from location $E078. Further, AIM 65 locations $FFFA and $FFFB contain $75 and $E0, respectively, so $\overline{\text{NMI}}$s cause the R6502 to fetch its next instruction from location $E075.

Figure 15-1 illustrates how the AIM 65's R6502 responds to an $\overline{\text{IRQ}}$ interrupt. Note that the return address (PCH and PCL) have been pushed onto the stack, and the Program Counter is "pointing to" location $E078, the location from which the R6502 will take its next instruction.

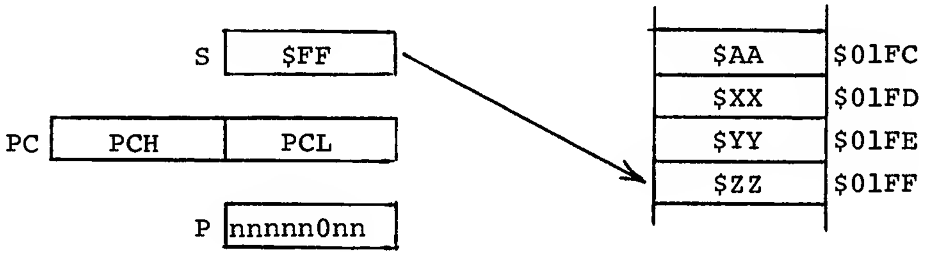
Instructions in an $\overline{\text{IRQ}}$ Interrupt Service Routine

Since an $\overline{\text{IRQ}}$ interrupt request automatically sets the Interrupt Disable bit (I) in the Processor Status Register -- to "lock-out" additional interrupt requests -- an $\overline{\text{IRQ}}$ interrupt service routine may clear the I bit immediately (thereby allowing "nested" interrupts) or leave it set until the present interrupt has been serviced.

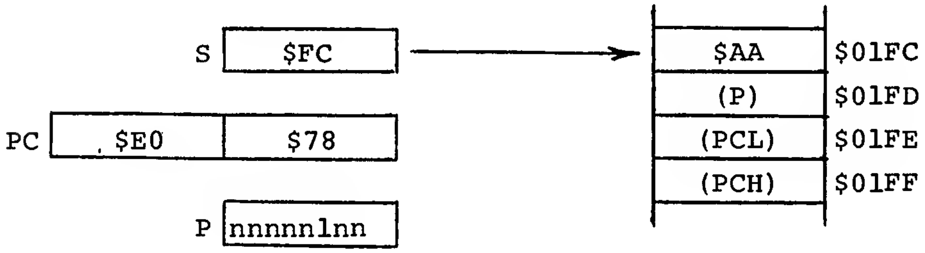
The only instruction that every interrupt service routine must have is:

INSTRUCTION	DESCRIPTION
RTI	Return from Interrupt

which is the return instruction for interrupt service routines, just as the RTS instruction is the return instruction for subroutines. The RTI instruction causes the Processor Status Register and Program Counter to be re-initialized with their pre-interrupt values from the stack. An RTI instruction will also enable $\overline{\text{IRQ}}$ interrupt requests, since the Processor Status Register's Interrupt Disable (I) bit was 0 when this register was pushed onto the stack.



Before $\overline{\text{IRQ}}$



After $\overline{\text{IRQ}}$

Figure 15-1. How the R6502 Responds to $\overline{\text{IRQ}}$

AIM 65 Interrupt Servicing

The AIM 65's printer, display and keyboard operate without interrupts (they are polled by the R6502), so the AIM 65 Monitor has no $\overline{\text{IRQ}}$ interrupt service routine. The AIM 65's R6502 CPU does have the facility to accept $\overline{\text{IRQ}}$ interrupts, however, from either the on-board, user-dedicated R6522 VIA or external peripheral I/O devices connected to the AIM 65, but the interrupt service routine must be provided by the user.

As you saw in Figure 15-1, when the R6502 accepts an $\overline{\text{IRQ}}$ interrupt, it loads the address $E078 into the Program Counter. What instruction does the R6502 find at address $E078? It finds the instruction JMP ($A400). This instruction directs the R6502 to jump to the address that is contained in $A400 (low byte) and $A401 (high byte), and continue executing at that indirect address.

The AIM 65 Monitor does not initialize locations $A400 and $A401. You must store the starting address of your interrupt service routine into those locations! For example, to use the interrupt capability of the on-board, user-dedicated R6522 VIA, you must store an

$\overline{\text{IRQ}}$ interrupt service routine for this device in memory, then store the starting address of this routine into locations $A400 and $A401. If your interrupt service routine starts at location $0300, you should store $00 into location $A400 and $03 into location $A401.

Figure 15-2 illustrates the sequencing of servicing an $\overline{\text{IRQ}}$ interrupt (perhaps from the R6522 VIA). The interrupt comes into the R6502 while the instruction STA 21 is being executed in the main program. Upon completion of this instruction, the R6502 accepts the interrupt and gets its next instruction from location $E078. Location $E078 contains the instruction JMP (A400), so the jump address is obtained from locations $A400 and $A401. Since these locations hold the address $0300, the jump is to location $0300--the starting address of the interrupt service routine. Because the interrupt could have occurred anywhere in the main program, the first instruction in the service routine is PHA, which save the accumulator on the stack. At the end of the interrupt service routine, PLA restores the accumulator (from the stack) and RTI causes the CPU to return to the next instruction in the main program, LDA #40.

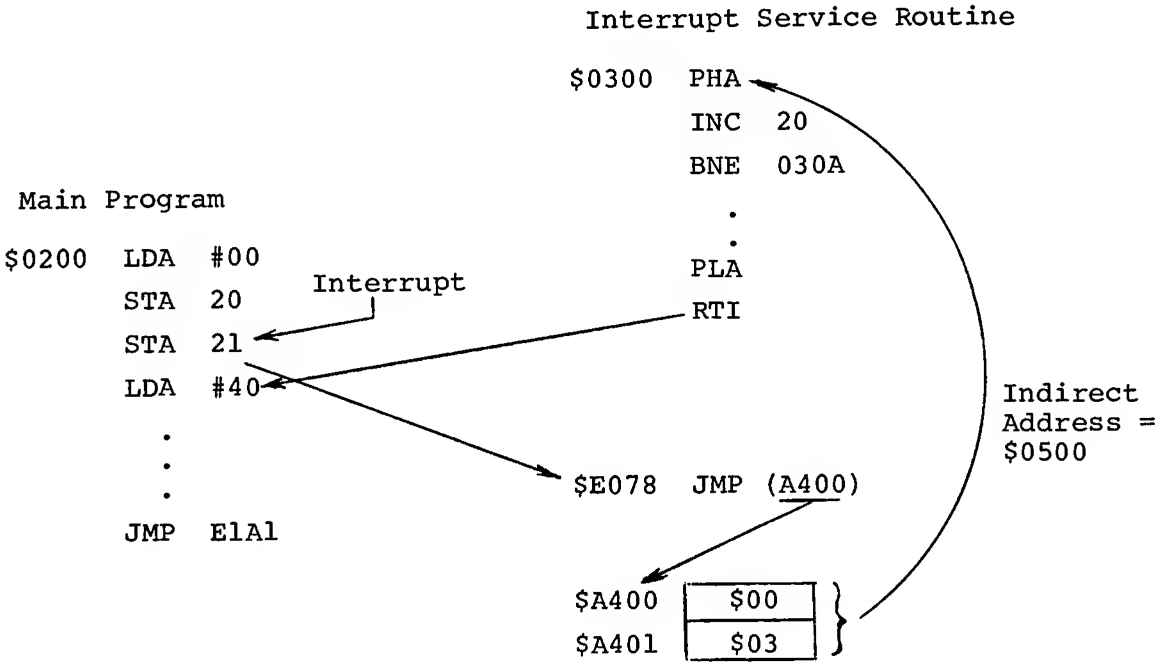


Figure 15-2. Example of AIM 65 Interrupt Servicing

The Laboratory Problem for This Experiment

Experiment 14 required you to develop a program that will measure how long the program runs until you press Reset. Now, in this experiment, you are to develop a program that does the same job, but rather than polling the VIA for a timeout, the program is to be set up so that a Timer 1 timeout generates an interrupt. A separate program, the interrupt service routine, will simply increment the time count and clear the Timer 1 Interrupt Flag, then return to the main program.

Programming the R6522 VIA for Interrupts

Since the basic problem here is the same as in Experiment 14, most of your previous VIA program instructions are usable in this experiment. Clearly, the values that are loaded into the Auxiliary Control Register and the Timer 1 Latch remain unchanged. Further, Timer 1 will still signal a timeout as before: by setting Bit 6 of the Interrupt Flag Register to 1.

What, then, needs to be done so that timeout of Timer 1 will cause an interrupt to the R6502 CPU? In order for Timer 1 to cause an interrupt, two conditions must be met:

1. The Interrupt Disable (I) bit in the Processor Status Register must have been pre-set to 1.
2. The Timer 1 Interrupt Enable bit in the VIA's Interrupt Enable Register ($A00E) must have been pre-set to 1.

The R6502 has two instructions which can be used to control the state of the Interrupt Disable (I) bit in the Status Register:

INSTRUCTION	DESCRIPTION
CLI	Clear Inerrupt Disable Bit
SEI	Set Interrupt Disable Bit

The first instruction CLI, clears I to 0, which will cause an external interrupt request to be serviced as soon as it is sensed by the CPU. The second instruction, SEI, sets I to 1, which will cause the CPU to ignore all subsequent $\overline{\text{IRQ}}$ interrupt requests.

Figure 15-3 shows the format of the Interrupt Flag Register and the Interrupt Enable Register. As you can see, the bit in the Interrupt Enable register that permits a Timer 1 timeout to generate an interrupt request is Bit 6. Note, too, that Bit 7 of both registers serves a special function. Bit 7 of the Interrupt Flag Register is set if any of the seven interrupt-causing conditions in the VIA have caused $\overline{\text{IRQ}}$ to be sent to the CPU. Bit 7 of the Interrupt Enable Register can be used to enable (Bit 7 = 1) or disable (Bit 7 = 0) selected VIA interrupt conditions, without affecting any other bits in the Interrupt Enable Register.

What value must be written into the Interrupt Enable Register to permit a timeout of Timer 1 to generate an interrupt request? Record your answer below.

Interrupt Enable Register ($A00E) =

For your convenience, Table 15-1 summarizes the memory locations that you will be using to access the R6522 VIA in this experiment.

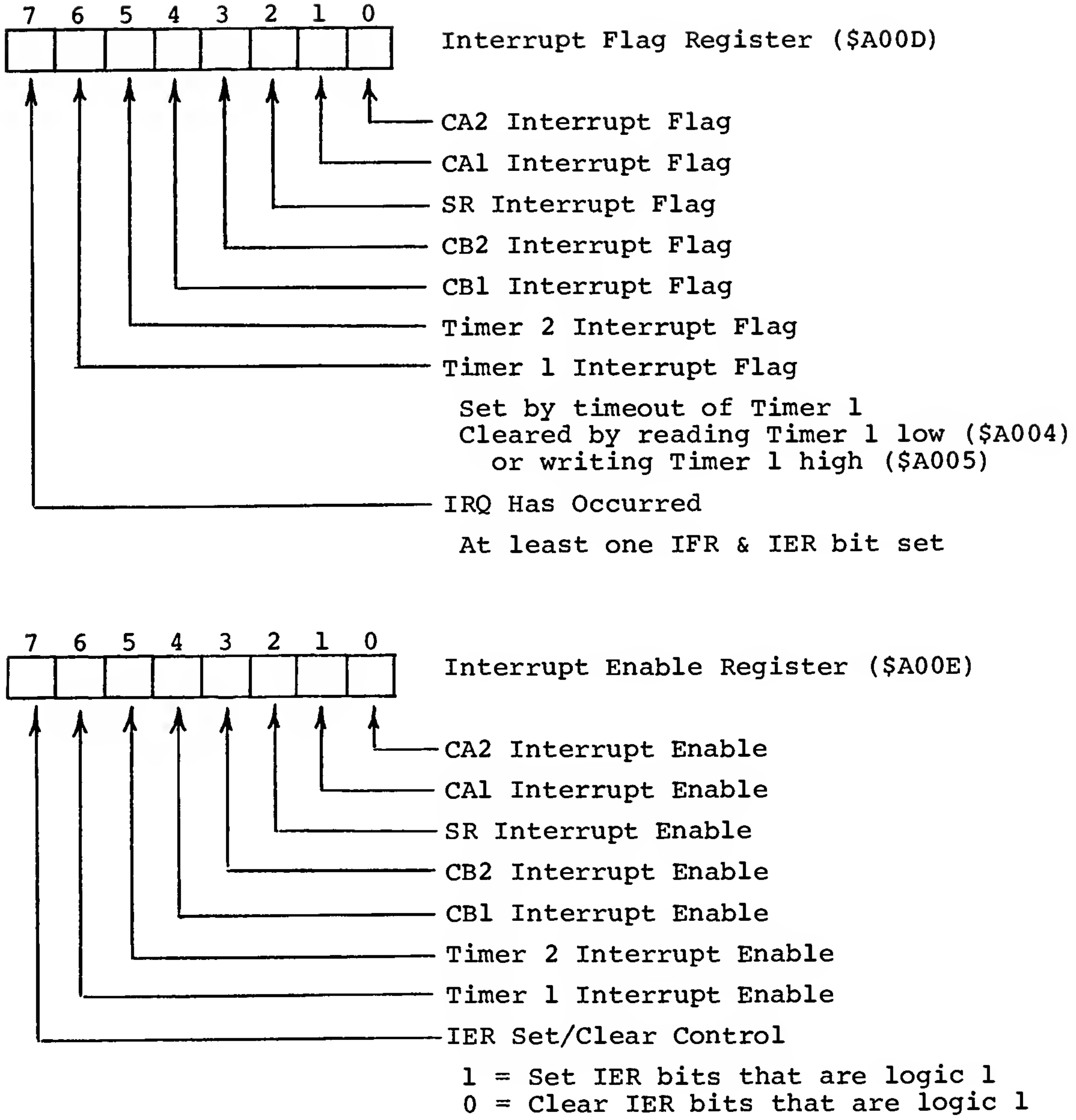

7 6 5 4 3 2 1 0
Interrupt Flag Register ($A00D)
CA2 Interrupt Flag
CA1 Interrupt Flag
SR Interrupt Flag
CB2 Interrupt Flag
CB1 Interrupt Flag
Timer 2 Interrupt Flag
Timer 1 Interrupt Flag
Set by timeout of Timer 1
Cleared by reading Timer 1 low ($A004)
or writing Timer 1 high ($A005)
IRQ Has Occurred
At least one IFR & IER bit set
7 6 5 4 3 2 1 0
Interrupt Enable Register ($A00E)
CA2 Interrupt Enable
CA1 Interrupt Enable
SR Interrupt Enable
CB2 Interrupt Enable
CB1 Interrupt Enable
Timer 2 Interrupt Enable
Timer 1 Interrupt Enable
IER Set/Clear Control
1 = Set IER bits that are logic 1
0 = Clear IER bits that are logic 1

Figure 15-3. VIA Interrupt Registers

Table 15-1 R6522 Memory Assignments for Experiment 15

Location	Register Selected	
	Write Operations	Read Operations
$A004	Write Timer 1 Latch low byte	Read Timer 1 Counter low byte Clear T1 Interrupt Flag (IFR6)
$A005	Write Timer 1 Latch high byte Clear T1 Interrupt Flag (IFR6) Initiate counting	Read Timer 1 Counter high byte
$A00B	Write Auxiliary Control Reg	Read Auxiliary Control Reg.
$A00D	Write Interrupt Flag Reg.	Read Interrupt Flag Reg.
$A00E	Write Interrupt Enable Reg.	Read Interrupt Enable Reg.

PROCEDURE

1. As in Experiment 14, you are to write a program that measures elapsed time until you press Reset on the AIM 65. The time must be accurate to 1/100 of a second (10 milliseconds). The time count, in memory, is incremented each time Timer 1 times out. Unlike Experiment 14, however, updating the time count must be performed by an interrupt service routine. Figures 15-4 and 15-5 are flowcharts for the main program and its interrupt service routine.

   Write the code for the main program in the space below. As in Experiment 14, use locations $20 and $21 to hold the time count. Further, your program should load $0300 into the interrupt vector (which means that the interrupt service routine must start at $0300).

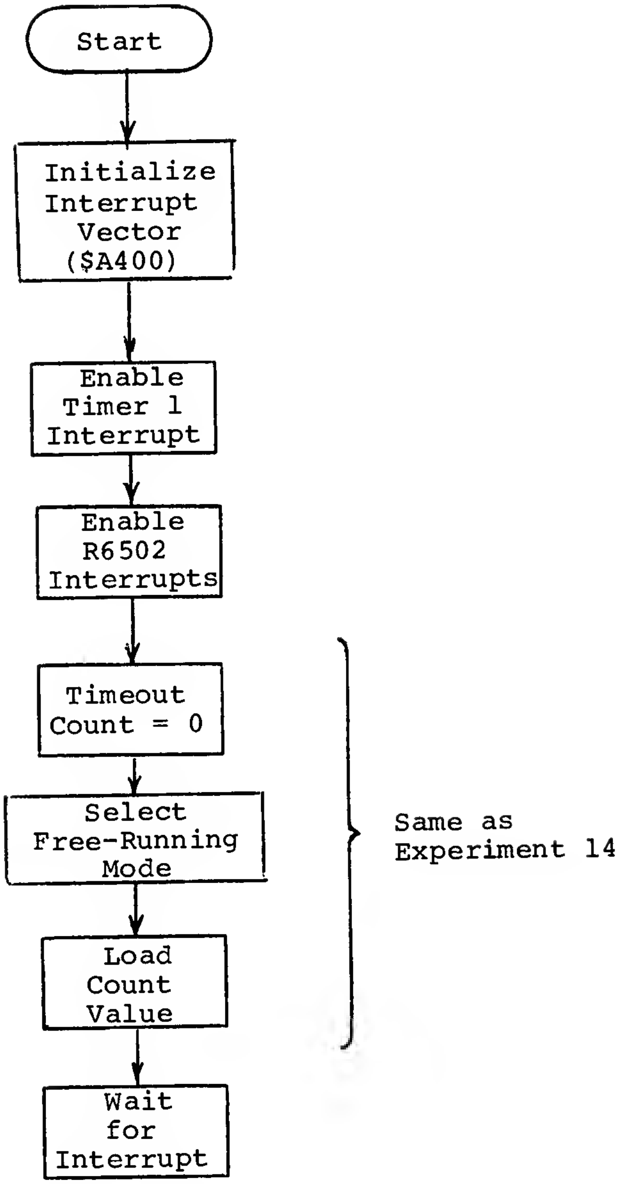

Start
Initialize Interrupt Vector ($A400)
Enable Timer 1 Interrupt
Enable R6502 Interrupts
Timeout Count = 0
Select Free-Running Mode
Load Count Value
Wait for Interrupt
Same as Experiment 14

Figure 15-4. Flowchart for Main Timing Program

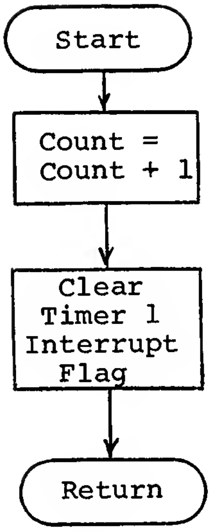


Figure 15-5. Flowchart for interrupt Service Routine

2. Write the code for the interrupt service routine in the space below.

3. Enter the main program and interrupt service routine into AIM 65. Put AIM 65 in the Run mode, then execute the program and press Reset after ten seconds have elapsed. Make three runs of ten seconds each, and record your results below.

"Correct" Count Value		Observed Count Values					
		Pass #1		Pass #2		Pass #3	
$20	$21	$20	$21	$20	$21	$20	$21
E8	03						

4. Does Timer 1 continue counting in the free-running mode after you have pressed Reset? To find out, examine its location in memory ($A00B) after a Reset, and record your observation below.

   Auxiliary Control Register ($A00B) =

   According to this value, in which mode is Timer 1 operating after a Reset? Answer below.

16

A TIMING PROGRAM WITH DECIMAL OUTPUT

OBJECT

To develop a more human-oriented timing program, using the R6522 VIA.

PRE-LAB PREPARATION

Read Section 2.2.1.3 in the R6500 Programming Manual. Review Experiments 12, 14 and 15.

DISCUSSION

The program you developed in Experiment 14, and modified in Experiment 15, provides a good introduction to the timing capabilities of the R6522 VIA, and shows one use of the VIA in a "real-life" application: measuring how much time has elapsed before a button is pressed. Programs of this type are common to many applications in which a computer-based product must interact with a human operator.

However, the program in Experiments 14 and 15 have two inherent disadvantages:

1. The answer in locations $20 and $21 is returned as a binary value, which can only be properly interpreted if you convert it to its decimal equivalent.
2. The answer remains in memory, requiring you to examine the contents of locations $20 and $21 to find out what result has been obtained.

In this experiment, you will modify the program in Experiment 15 so that it maintains the 0.01-second time count in Binary Coded Decimal (BCD) form, and continuously displays the "seconds" portion of the count.

Maintaining a Decimal (BCD) Time Count

As discussed in the Rockwell literature, and in Experiment 2 of this manual, the R6502 can operate in either of two arithmetic modes, binary or decimal, depending on the state of the Decimal Mode (D) bit in the Processor Status Register. When D = 0, the R6502 performs binary add and subtract operations; when D = 1, the R6502 performs decimal add and subtract operations. The two instructions that select the arithmetic mode are SED (Set

Decimal Mode) and CLD (Clear Decimal Mode).

In Experiments 14 and 15, the time count was maintained as a binary number, so increment (INC) instructions could be used to increase the count. However, these instructions can perform only a binary increment, so a decimal time count must be increased with an add (ADC) sequence.

Converting the Decimal Time Count

From Experiments 12 and 13 you realize that binary or BCD data cannot be directly output to the AIM 65 display--it must first be converted to its ASCII representation. What is involved in converting a two-digit BCD number in location $21 to ASCII? You have already received a good start in the right direction, with the discussion of "Binary to ASCII-Based Hexadecimal" conversion in Experiment 12. This discussion even includes a flowchart for the conversion subroutine, as Figure 12-1.

You will be pleased to know that converting BCD values to ASCII-based decimal is similar to, but easier than, converting binary values to ASCII-based hexadecimal. Why? Because, with BCD values, your conversion subroutine does not need to check whether a four-bit data value is greater than 9; if the value is BCD, it will always be less than or equal to 9! Therefore, the ASCII representation of a BCD digit is obtained by adding $30 to the digit.

Displaying the Decimal Time Count

Once the interrupt service routine has been properly modified to maintain the count in BCD form, locations $20 and $21 will each hold two BCD digits, of four bits each. In order to display the digits in either of these bytes, the digits must be "unpacked", converted to ASCII, then output to the display. Naturally, the high-order digit must be processed first, so that it appears in the left most position on the display.

To avoid getting "bogged down" manipulating the display, we will use two display subroutines in the AIM 65 Monitor: CLR (entry address $EB44), which clears the display, and OUTPUT (entry address $E97A), which outputs the ASCII character in the Accumulator to the AIM 65's display and printer.

PROCEDURE

1. You are to develop an enhanced version of the elapsed-time-until-Reset program that was introduced in Experiment 14 and modified (for interrupts) in Experiment 15. The program here is similar to the one in Experiment 15, except that the time count in locations $20 and $21 is to be maintained in BCD form, rather than decimal form. This stipulation will not affect the main program, but it will require you to develop a new interrupt service routine in which the time count is updated with add (ADC) operations--in Decimal Mode--instead of with increment (INC) operations. Figure 16-1 is a flowchart for the BCD-based interrupt service routine.

   Write the code for the new interrupt service routine in the space below.

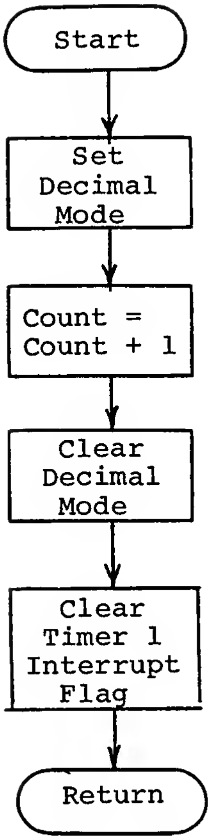

Start
Set Decimal Mode
Count = Count + 1
Clear Decimal Mode
Clear Timer 1 Interrupt Flag
Return

Figure 16-1. Flowchart for BCD-Based Interrupt Service Routine

2. Enter the main program (from Experiment 15) and the interrupt service routine into AIM 65. Put AIM 65 in the Run mode, then execute the program after ten seconds have elapsed. Make three runs of ten seconds each, and record your results below.

Count Value for 10 Secs.		Observed Count Values					
		Pass #1		Pass #2		Pass #3	
$20	$21	$20	$21	$20	$21	$20	$21
00	10						

3. With the time count maintained in decimal, rather than binary, what is the maximum time, in seconds, that can be properly recorded in locations $20 and $21? Answer below.

Maximum total time (seconds) =

4. With the time count being maintained in decimal form, we can now add the instructions to have the seconds count--the contents of location \$21--displayed. Figure 16-2 is a flowchart showing the types of instructions that must be added to the interrupt service routine for display. Following this flowchart, write the code for your modified interrupt service routine below.

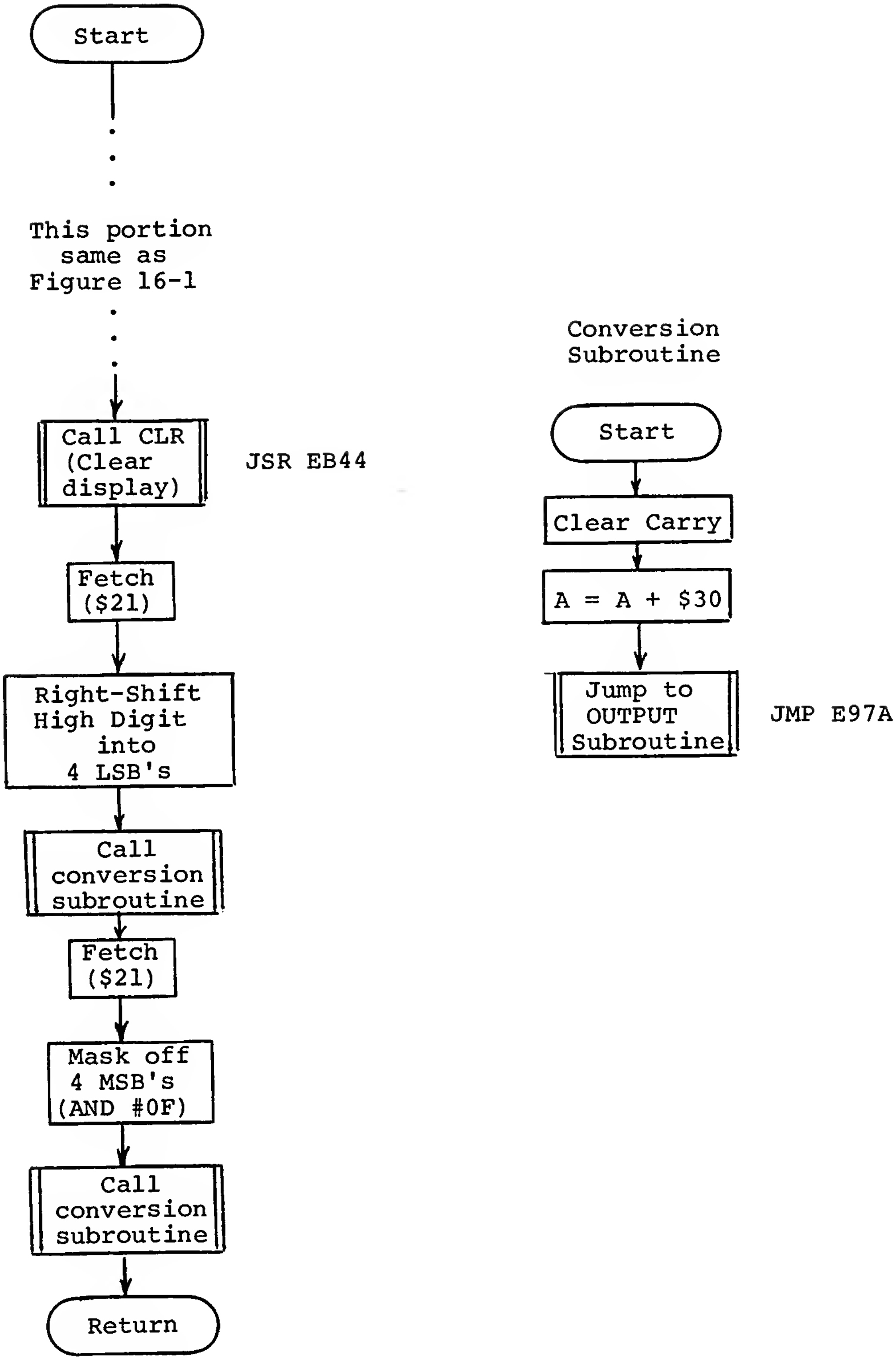

Start
This portion
same as
Figure 16-1
Call CLR
(Clear
display)
JSR EB44
Fetch
($21)
Right-Shift
High Digit
into
4 LSB's
Call
conversion
subroutine
Fetch
($21)
Mask off
4 MSB's
(AND #0F)
Call
conversion
subroutine
Return
Conversion
Subroutine
Start
Clear Carry
A = A + $30
Jump to
OUTPUT
Subroutine
JMP E97A

Figure 16-2. Interrupt Service Routine With Display

5. Turn the AIM 65 printer off (with CTRL/PRINT), and run the program with your new, display-producing interrupt service routine.

17

THE AIM 65 ASSEMBLER

OBJECT

To learn how to prepare programs using the AIM 65 Assembler.

PRE-LAB PREPARATION

Read Sections 4 and 5 of the AIM 65 User's Guide for descriptions of the AIM 65 Text Editor and Assembler, respectively.

REQUIRED LAB EQUIPMENT

AIM 65 with 4K bytes of RAM memory and Assembler.

DISCUSSION

Until now, all of your programs have been entered into AIM 65 memory in the so-called "mnemonic instruction entry mode;" that is, using the I command. This method is satisfactory for small programs, but has some inherent drawbacks for programs that are long or somewhat complex. For example, the I command forces you to enter all operands in hexadecimal form, and does not accept decimal, binary or ASCII operands. Further, the I command does not allow you to annotate your programs with comments, nor does it allow you to reference constants or instructions with symbolic labels (operand addresses must be absolute, like JMP 421F, rather than symbolic, like JMP THERE). The optional AIM 65 Assembler eliminates these inconveniences, and permits programs to be entered in about the same form that you write them on paper. That is, the Assembler translates source code (programs written with labels, comments and so on) to object code (binary opcodes and operands) that can be executed by the R6502 CPU.

The AIM 65 Text Editor

If the I command cannot be used to enter Assembler source code into memory, how do we put this source code into the AIM 65? We enter source code by using a special AIM 65 program called the Text Editor. The Text Editor will allow you to assign a portion of AIM 65 read/write memory to act as a Text Buffer, an area into which you can enter your program instructions, comments and labels. (The Text Buffer merely collects information typed in from the keyboard, and stores it in ASCII form. It makes no qualitative decisions as to whether or not the program contains errors.) Once the Text Buffer has been established you can use Text Editor commands to:

- Enter from and exit to the AIM 65 Monitor
- Input, output and update (add, delete or change) the text
- Select the active line in the Text Buffer
- Find and change character strings

The Text Editor commands are summarized in Table 17-1.

Makeup of an Assembler Source Code Program

We have just described two ways in which Assembler source code (in the Text Buffer) can differ from the mnemonic programs you have been entering with the I command: source code programs can contain comments (and will, if you are a thorough programmer) and can contain symbolic labels and symbols. The source code program can also include special instructions to the Assembler, called Assembler directives. There are nine Assembler directives; they are described in Section 5.8 of the AIM 65 User's Guide. For the moment, let us discuss two of these directives that must be included in every source code program, one preceding the first program instruction and another following the last instruction in the program.

The Assembler directive that precedes the program is an eouate directive, which is denoted with an equal sign (=). In this particular application, the equate directive is used to specify the starting address of the program in memory, the first location at which the object code is to be stored. This starting address will be specified with a statement of the form

```
*=$aaaa
```

where "aaaa" represents the four hexadecimal digits of the program's starting address. For example, if you want the object code to be loaded starting at memory location $0200, your source program would be preceded with the equate directive *=$0200. Note that this step is the Assembler equivalent of initializing the Program Counter (with a * Monitor command) when you enter programs under the I command.

The required Assembler directive at the end of a program is, appropriately enough, a .END directive. This directive informs the Assembler to ignore the remaining information in the Text Buffer.

Other Assembler Directives

The directives just described, equate and .END, are only two of the nine directives that can be used in Assembler source programs. All nine directives are described in Section 5.8 of the AIM 65 User's Guide. Their descriptions will not be repeated here, but it will be worthwhile, for purposes of this experiment, to discuss one of the other directives, .BYTE, and to discuss another use for the equate directive.

The .BYTE directive is used to store eight-bit data values in consecutive locations in memory, and is primarily employed in setting up data tables. A .BYTE directive can have up to 20 operands, which can be in hexadecimal ($ prefix), decimal (no prefix), binary (% prefix) or ASCII (enclosed by ') form.

Hexadecimal, decimal and binary operands must be separated by commas. For example, the statement

Table 17-1. AIM 65 Text Editor Commands
(Courtesy of Rockwell International)

ENTER AND EXIT EDITOR COMMANDS

E — Enter and Initialize Editor
<E>
EDITOR
FROM = [ADDRESS] TO = [ADDRESS]
IN = [INPUT DEVICE]
Note: Defaults are TO = $0200,
FROM = Last contiguous RAM, IN = Keyboard

Q — Exit the Text Editor and Return to Monitor
= <Q>

LINE ORIENTED COMMANDS

R — Read Lines into Text Buffer from Input Device
= <R>
IN = [INPUT DEVICE]

I — Insert One Line of Text Ahead of Active Line
= <I>
INSERTED TEXT LINE
ACTIVE LINE OF TEXT

K — Delete Current Line of Text
= <K>
DELETED LINE OF TEXT
ACTIVE LINE OF TEXT

U — Move the Text Pointer Up One Line
= <U>
PRIOR LINE OF TEXT

D — Move the Text Pointer Down One Line
= <D>
NEXT LINE OF TEXT

T — Move the Text Pointer to the Top of the Text
= <T>
TOP LINE OF TEXT

B — Move the Text Pointer to the Bottom of the Text
= <B>
BOTTOM LINE OF TEXT

L — List Lines of Text to Output Device
= <L>
/[DECIMAL NUMBER]

SPACE — Display the Active Line
= < >
ACTIVE LINE OF TEXT

STRING ORIENTED COMMANDS

F — Find a Character String
= <F>
[CHARACTER STRING]
LINE CONTAINING CHARACTER STRING

C — Change a Character String
= <C>
[OLD STRING]
LINE CONTAINING OLD STRING
TO = [NEW STRING]
SAME LINE, WITH NEW STRING

```
DTABL    .BYTE    $2F,32,%01110100
```

stores the binary equivalent of $2F_{16}$, 32_{10} and 01110100_2 into symbolic locations DTABL, DTABL+1 and DTABL+2, respectively.

ASCII operands can be represented as a string of alphanumeric characters enclosed by single quotes. For example, the statement

```
ATABL   .BYTE    'NAME '
```

stores the ASCII codes (refer back to Figure 11-1) for the characters N, A, M, E and SPACE into the five memory locations that start at ATABL.

Our previous discussion of the equate directive (=) described its one required application, to specify the starting address of the program. Another common application of this directive is in assigning a value to symbols that will be used in the program. A symbol is a one- to six-character name that can represent a constant data value or an absolute address. Symbols provide two distinct advantages:

1. They make a program listing easier to follow, because data and address values are identified with (hopefully) meaningful names rather than numeric digits.
2. They make programs easier to change. You can change a symbol's value throughout a program by simply changing the equate statement in which the value is assigned.

For example, the equate statement

```
CONS=16
```

will cause the Assembler to substitute decimal 16 for each occurrence of the symbol CONS in the program. That is, the instruction LDA #CONS will be converted into the binary representation of the instruction LDA #16.

Similarly, the equate statement

```
COMIN=$E1A1
```

will cause each occurrence of the instruction JMP COMIN to have the same effect as the by-now-familiar instruction JMP E1A1 had in your programs in previous experiments (i.e., to return control to the Monitor's COMIN routine at the conclusion of a program).

Entering Source Code into the Text Buffer

Now that we know what the Text Buffer is used for, and what kind of information is to be put into it, we are ready to start using it. To establish a Text Buffer, you simply press the E key. AIM 65 will respond by displaying the message EDITOR, and then ask you to enter the starting and ending limits of the Buffer, and the origin of the input, with the prompts FROM=, TO= and IN=, respectively. If you press the space bar in response to each of these prompts, you will establish a Text Buffer from location $0200 to the last location in RAM (location $1000, if you have a 4K-byte system) and select the AIM 65 Keyboard as the input device.

With the Text Buffer established, you can enter the program (and the comments and

Assembler directives) as a series of lines of text, ending each line by pressing RETURN. The information is usually entered in this order:

1. Equate directives to assign values to symbols in the program.
2. The equate directive that specifies the starting address, or origin, of the object code; for example, *=$200.
3. Program instructions and comments (a comment should precede the instruction to which it applies).
4. Data tables and messages constructed with .BYT, .WOR and .DBY directives.
5. The .END directive that terminates the program information.

When program entry is complete, or you want to return to the Monitor at any other time, press ESC. The Text Editor can be re-entered at the top of the Text Buffer by pressing the T key.

The AIM 65 Assembler

As you certainly know by now, the program in the Text Buffer cannot be executed directly by the R6502 CPU; before it can be executed, all mnemonic instructions must be translated into binary form, all labels must be replaced by absolute addresses and the operands of all directives must be converted into binary values in memory. The program that performs these tasks is called the Assembler.

When executed, the Assembler will make two passes through the program that resides in the Text Buffer. During Pass 1, the Assembler calculates the absolute address that corresponds to each label in the program and uses this information, along with the binary value of each symbol in the program, to build a Symbol Table in memory. The Assembler uses this Symbol Table during Pass 2 to convert the symbolic source program in the Text Buffer to an object code program, and stores that object code in the portion of memory assigned by the equate directive in your source program. During Pass 2, the Assembler also checks the program for syntax errors--invalid mnemonics, undefined labels and so on--and prints a code (see Table 5-1 in Section 5.4 of the AIM 65 User's Guide) for each error it encounters.

Assembling a Program

Table 17-2 summarizes the sequence of prompts that will be displayed after you initiate the Assembler with the N key.

The prompts, and your responses, are as follows:

PROMPT	RESPOND WITH
FROM =	Four-digit starting address (hex) of Symbol Table, then press RETURN.
TO=	Four-digit ending address (hex) of Symbol Table, then press RETURN.
IN=	Letter code of device containing source program. Enter M for memory.

LIST?	Enter Y for listing of program and errors, or N for listing of only errors.
LIST-OUT=	Letter code of listing device. Enter P for Printer.
OBJ?	Enter N to store object code in memory.

Table 17-2. Sequence of Operations for the Assembler
(Courtesy of Rockwell International)

```
<N>
ASSEMBLER
FROM = [ADDRESS]TO = [ADDRESS]
IN = [INPUT DEVICE]
LIST?[Y, N]
LIST-OUT = [OUTPUT DEVICE]
OBJ?[Y, N]  Note: N = Object code to Memory
OBJ-OUT = [OUTPUT DEVICE]  Note: Prompts only on Y response to OBJ?
PASS 1
 SYM TBL OVERFLOW  }
 ASSEMBLER         }  Displayed only if Symbol Table overflows
PASS 2
= = AAAA LABEL                     }        Displayed only if
OBJECT CODE   MNEMONIC OPCODE      }        LIST?Y, or LIST?N
SYMBOLIC OPERAND   ;COMMENT        }        and error detected
* * ERROR NN  Note: Error code displayed only on error
ERRORS =  MMMM  Decimal count of errors detected
```

The AIM 65 will give you a visual indication of the Assembler's progress, by displaying PASS 1 while Pass 1 is being executed and PASS 2 while Pass 2 is being executed. During Pass 2, you will receive an assembly listing of the program, along with an error code for each line containing an error. If your program has an error, press the T key (to reenter the Text Editor) and fix the offending line(s) using the Text Editor commands. When all errors have been corrected, reassemble the program with the N key. At the completion of the assembly, you can run your program as you did in all of the preceding experiments--by initializing the Program Counter with the * command (initialize it to the starting address of the object code) and then pressing the G key.

Memory Usage With the Assembler

In the preceding discussion, we have described how three separate portions of AIM 65 RAM memory must be allocated when you use the Assembler. One portion holds the source code (in the Text Buffer), another holds the Symbol Table that the Assembler generates during Pass 1, and a third portion holds the object code for the program. You must make certain that none of these three portions of memory overlap. Figure 17-1 is a memory map showing the source code for a program occupying locations $0200 through $040E, the object code occupying locations $0500 through $063C and the Symbol Table occupying locations $0800 through $0840.

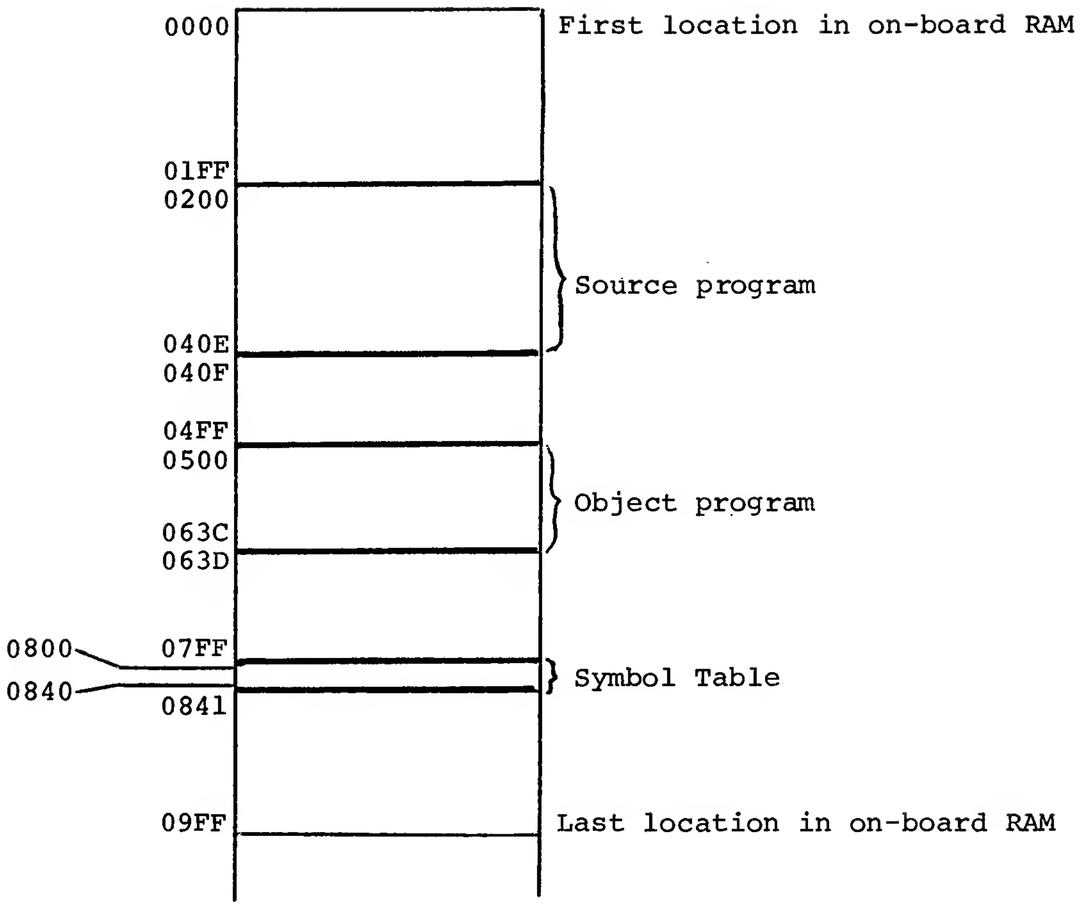

0000
First location in on-board RAM
01FF
0200
Source program
040E
040F
04FF
0500
Object program
063C
063D
0800
07FF
Symbol Table
0840
0841
09FF
Last location in on-board RAM

Figure 17-1. Memory Map of Source, Object and Symbol Table

PROCEDURE

1. In Experiment 4, you developed a program to arrange two single-byte numbers in memory (locations $20 and $21) in increasing order. In this experiment we will go through all the steps needed to enter that program and run it using the AIM 65 Text Editor and Assembler. Following is the symbolic source code you will be working with in the present experiment.

```
       LDA  $21
       CMP  $20
       BCC  SWAP
       BEQ  EQUAL
       LDA  #0
       JMP  STORE
EQUAL  LDA  #1
       JMP  STORE
SWAP   LDX  $20
       STX  $21
       STA  $20
       LDA  #2
STORE  STA  $22
       JMP  $E1A1
```

2. Before entering this program into memory, you must set up the Text Buffer, using the Editor's E command. Press E, and allocate the Text Buffer to memory between $0200 and $040E. Record your responses to the E command's prompts in the spaces below.

EDITOR

FROM= TO=

IN=

3. Enter the source program (Step 1) into the Text Buffer. The source program must be preceded by an equate directive that specifies where the object program is to be stored; you are to specify the address $0500. The source program must be followed with a .END directive.

4. After entering the .END directive, press ESC, to leave the Editor and return to the Monitor. Your source program is now in the Text Buffer, in ASCII form. To generate the object program, the source must be assembled, with the use of the N command. Press N and record your response in the spaces below. Specify addresses $0800 and $1000 as the Symbol Table limits.

ASSEMBLER

FROM= TO=

IN=

LIST?

LIST-OUT=

OBJ?

5. If there are errors, you did not follow the preceding instructions correctly. Any existing errors must be fixed by modifying the source program (in the Text Buffer). Which key would you press to enter the Text Buffer without re-initializing it? Answer below.

   To re-enter the Text Buffer, press ________

6. If you have errors, go back and correct them. If not, use the K command to disassemble your object code, starting at location $0500.

7. Now initialize ($20) = $96 and ($21) = $0E, and run your program with the G command. Record the results below.

   (20) = ($21) = ($22) =

   If the contents of location $22 agree with the results you obtained in Step 2 of Experiment 4, your work has been successful.

ANSWERS TO EXPERIMENTS

The following pages provide answers to the questions contained in the Discussion portion of the experiments, as well as solutions to the problems assigned in the Procedures. All answers given are those proposed by the author. Some of your answers--particularly programs and flowcharts--may differ from those printed here without necessarily being "incorrect." Since this is an educational manual, the best solution is one that not only solves the problem correctly, but helps you learn the principles being studied.

All programs are listed in the form you would enter them into AIM 65 (i.e., with three-letter mnemonics and hexadecimal operands) except that here: (1) the instructions are often followed by comments, to explain what is being done, and (2) most Branch, JMP and JSR instructions have a label as the operand. Unless you have an assembler, using labels is clearly "cheating." However, we have used labels here to help you follow the flow of the program, and perhaps help you to relate a certain sequence in the answer program to the equivalent sequence in your own solution.

EXPERIMENT 1. GETTING TO KNOW THE AIM 65

1. After you press Reset,

   PC = $0000 A = $00

2. You should have used the R command (Display Register Values)

3. At this point, location $40 contains a random value, so anything you wrote down is "correct", as long as it accurately reflects what $40 contains.

4. Use the command M=40 to display the contents of location $40.

5. The correct program is:

```
LDA  #AC
STA  40
```

6. LDA #AC is using the Immediate addressing mode.

7. STA 40 is using the Zero Page addressing mode. The Absolute addressing version of this instruction, STA 0040, will also work, but it's less efficient when addressing locations in Zero Page.

8. The Program Counter will point to $0200 if you use the command

```
*=0200
```

9. The I command (Enter Mnemonic Instruction Entry Mode) is used.

10. At this point, the display shows

```
0200 ^
```

   which indicates that the Program Counter is pointing to location $0200. AIM 65 is waiting for you to enter an instruction at that address.

12. Using the R command, you discover that

   PC = $0204

   It has this value because the two-instruction program occupies four locations in memory, $0200 through $0203, so the Program Counter has advanced to the location of the next instruction (if there had been one), $0204.

13. Execute the program with

```
G/02
```

   NOTE: This will not work correctly if your AIM 65 is switched to the Run mode!

EXPERIMENT 1 (CONT'D)

14. At this point

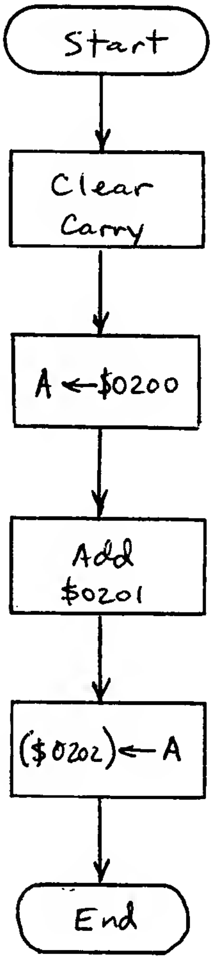
PC = \$0204   A = \$AC   \$40 = \$AC

15. For the program listed in Step 5, the information is:

LOCATION	HEX. CONTENTS	DESCRIPTION OF CONTENTS
\$0200	\$A9	Op code for LDA (Immediate)
\$0201	\$AC	Immediate operand
\$0202	\$85	Op code for STA (Zero Page)
\$0203	\$40	Zero page address

EXPERIMENT 2. ADDITION OPERATIONS

1. The flowchart looks like this:


Start
Clear Carry
A ← $0200
Add $0201
($0202) ← A
End

The program to perform the add operation is:

```
CLC              Clear Carry
LDA   0200       Load $0200
ADC   0201       Add $0201 to it
STA   0202        and store the result
```

The results for the test cases are:

$0200		$0201		$0202		Carry? (Yes/No)
Dec.	Hex.	Dec.	Hex.	Dec.	Hex.	
143	8F	131	83	274	12	Yes
46	2E	65	41	111	6F	No
59	3B	197	C5	256	00	Yes

The answers are invalid in the first and third case, because a single memory location cannot hold a value that exceeds 255.

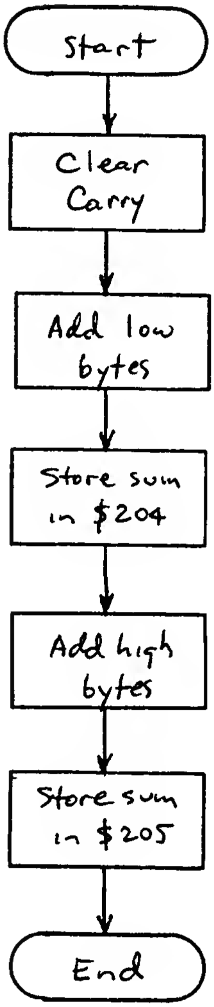
2. The program to add signed numbers will be no different than the program to add unsigned numbers. To the R6502, both types of numbers are simply values in memory; it's up to you to interpret what kind of numbers are being added.

$0200		$0201		$0202		Overflow? (Yes/No)
Dec.	Hex.	Dec.	Hex.	Dec.	Hex.	
106	6A	89	59	195	C3	Yes
127	7F	3	03	130	82	Yes
32	20	-3	FD	29	1D	No
-90	A6	-62	C2	-152	68	Yes

All three answers that have overflow are invalid, because a single memory location cannot hold a signed value that exceeds +127 or -128.

EXPERIMENT 2 (CONT'D)

3. The flowchart for the double-precision signed add looks like this:


Start
Clear Carry
Add low bytes
Store sum in $204
Add high bytes
Store sum in $205
End

The program is:

```
CLC               Clear Carry
LDA   0200        Add low bytes
ADC   0202
STA   0204         and store the result
LDA   0201        Add high bytes
ADC   0203
STA   0205         and store the result
```

EXPERIMENT 2 (CONT'D)

The answers for the test run are:

Addend #1 $0200,$0201	Addend #2 $0202,$0203	Sum $0204,$0205	Overflow? (Yes/No)
$2653	$41AB	$670E	No
$83C7	$F1B4	$757B	Yes

4. You can determine whether a carry occurred between bytes by running the program in sections, and examining the state of Carry after each section. For example, for the program in Step 3, you could run the first four instructions (which add the low bytes) and examine Carry after that run.

EXPERIMENT 3. SUBTRACTION AND LOGICAL OPERATIONS

1. 24-bit subtraction program.

```
SEC            Set Carry for subtract
LDA  0203      Subtract low-order bytes
SBC  0200
STA  0203       and store result
LDA  0204      Subtract middle bytes
SBC  0201
STA  0204       and store result
LDA  0205      Subtract high-order bytes
SBC  0202
STA  0205       and store result
```

2. Results of subtracting \$74A017 from \$267BFE

Subtrahend			Minuend		
\$0200	\$0201	\$0202	\$0203	\$0204	\$0205
17	A0	74	FE	7B	26

Results			
\$0203	\$0204	\$0205	Carry =
E7	DB	B1	0

By checking the final state of Overflow, find out whether the result is valid. V = 0 here, so the result is valid.

3. To find out how many of the three subtractions, you can run the program in the Step mode in three steps: execute the first four instructions, then the next three, then the final three. Borrow is indicated by the state of the Carry after each execution.

For the numbers in Step 2, borrows were generated by the middle-order and high-order subtractions.

Total number of borrows = 2

EXPERIMENT 4. PROGRAM SEQUENCING

1. Here is a program that follows the flowchart in Figure 4-1:

```
        LDA  21       Load first value
        CMP  20       Values out of order?
        BCC  SWAP
        BEQ  EQUAL    No.  $20 is
        LDA  #00       less than $21
        JMP  STORE
EQUAL   LDA  #01       or equal to $21
        JMP  STORE
SWAP    LDX  20       Yes.  $20 is greater than $21
        STX  21       Exchange $20 and $21
        STA  20
        LDA  #02
STORE   STA  22       Store indicator in $22
        JMP  E1A1      and return to Monitor
```

   Incidentally, we could increase the efficiency of this program by replacing both JMP STORE instructions (which occupy 3 bytes in memory and take 3 cycles to execute) with Branch instructions (which occupy 2 bytes in memory and take 3 cycles to execute). Due to the construction of the program, the first JMP STORE could be replaced with BNE STORE and the second JMP STORE could be replaced with BEQ STORE.

2. For the test runs, the contents of $22 are $02, $01, $00 and $01, respectively.

3. The flowchart for the maximum value program is shown on the next page.

   The code for the maximum value program is:

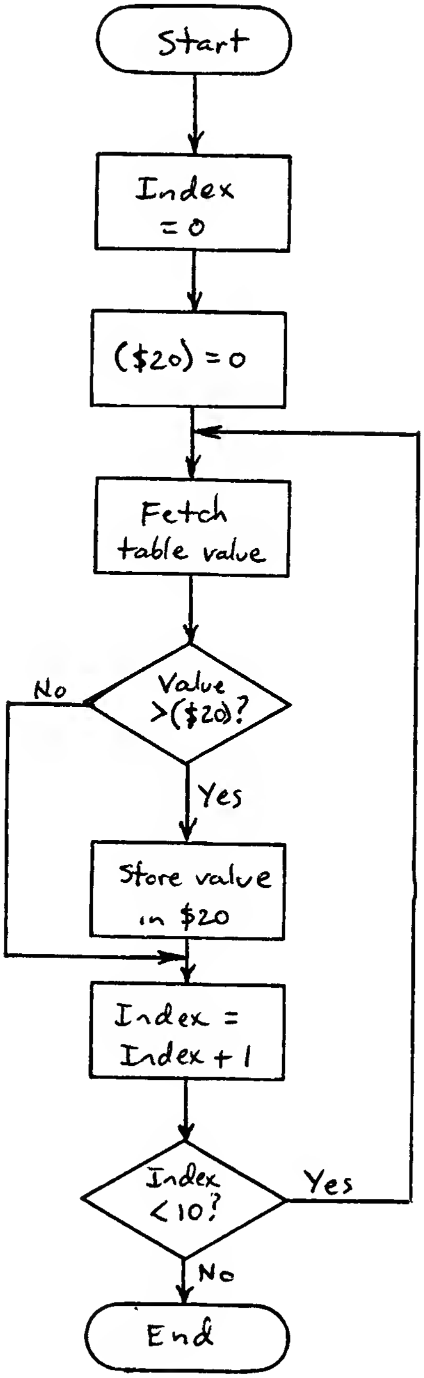
```
        LDX  #00      Index = 0
        STX  20       Result location = 0
LOOP    LDA  21,X     Fetch next table value
        CMP  20       Is it greater than ($20)?
        BCC  UPIND
        BEQ  UPIND
        STA  20       Yes.  Change location $20
UPIND   INX           Increment index
        CPX  #10      Finished with table?
        BCC  LOOP     No. Go back for next value
        JMP  E1A1     Yes.  Return to Monitor
```

4. If the program works correctly,

   Result in $20 = $FA

EXPERIMENT 4 (CONT'D)

3. The flowchart for the maximum value program is:


Start
Index = 0
($20) = 0
Fetch table value
Value >($20)?
No
Yes
Store value in $20
Index = Index + 1
Index <10?
Yes
No
End

EXPERIMENT 6. MULTIPLICATION OPERATIONS WITH SHIFT & ROTATE

1. The results should be the same as shown in the Discussion. That is, ASL A should yield $68, LSR A should yield $1A, ROL A should yield $69 and ROR A should yield $9A.

2. The flowchart for the multiplication is shown on the next page.

   The code for the multiplication program should look like this:

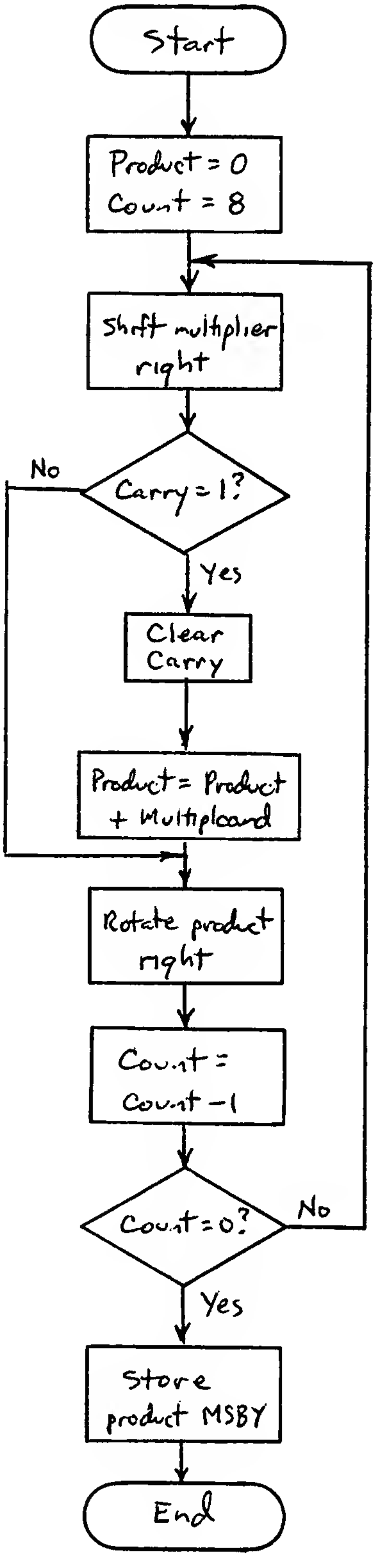
```
       LDA  #00     Clear MSBY of product
       LDX  #08     Multiplier bit count = 8
NXTBT  LSR  20      Get next multiplier bit
       BCC  ALIGN   Multiplier bit = 1?
       CLC          Yes.  Add multiplicand
       ADC  21
ALIGN  ROR  A       Rotate partial product right
       ROR  22
       DEX          Decrement bit count
       BNE  NXTBT   Loop until 8 bits are done
       STA  23       then store product MSBY
       JMP  E1A1
```

3. The results for the four test cases are:

Multiplier $20		Multiplicand $21		Product $22,$23	
Dec.	Hex.	Dec.	Hex.	Dec.	Hex.
255	FF	1	01	255	00FF
103	67	124	7C	12772	31E4
255	FF	0	00	0	0000
255	FF	255	FF	65025	FE01

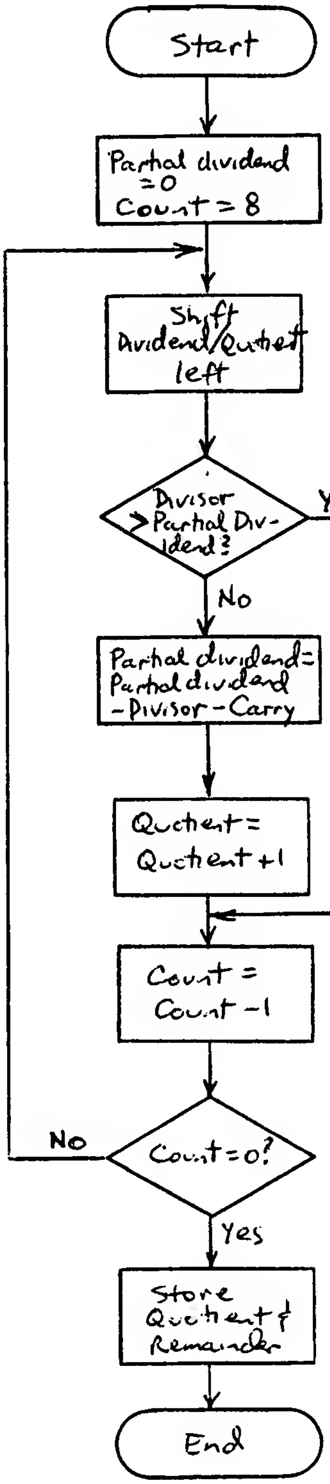
EXPERIMENT 6 (CONT'D)

2. The flowchart for the single-precision multiplication is:


Start
Product = 0
Count = 8
Shift multiplier right
Carry = 1?
No
Yes
Clear Carry
Product = Product + Multiplicand
Rotate product right
Count = Count − 1
Count = 0?
No
Yes
Store product MSBY
End

EXPERIMENT 7. DIVISION OPERATIONS

1. The flowchart for the single-precision division is:


Start
Partial dividend =0
Count = 8
Shift Dividend/Quotient left
Divisor > Partial Dividend?
Yes
No
Partial dividend= Partial dividend -Divisor -Carry
Quotient= Quotient +1
Count = Count -1
Count=0?
No
Yes
Store Quotient & Remainder
End

EXPERIMENT 7 (CONT'D)

The program code for the division is:

```
        LDA  #00      Clear partial dividend
        LDX  #08      Dividend bit count = 8
NXTBT   ASL  21       Shift dividend/quotient left
        ROL  A         into partial dividend
        CMP  20       Compare divisor to partial dividend
        BCC  CNTDN    Divisor greater than partial dividend?
        SBC  20       No.  Subtract divisor
        INC  21        and set bit in quotient
CNTDN   DEX           Decrement bit count
        BNE  NXTBT    Loop until 8 bits are done
        STA  22        then store remainder
        JMP  E1A1
```

Note that there was no need to set the Carry prior to the subtract, since it had to be set for BCC CNTDN to "fall through" to the subtract operation.

2. The results of the four test cases are:

Divisor $20		Dividend $21		Quotient $21		Remainder $22	
Dec.	Hex.	Dec.	Hex.	Dec.	Hex.	Dec.	Hex.
1	01	255	FF	255	FF	0	00
15	0F	170	AA	11	0B	5	05
255	FF	1	01	0	00	255	FF
254	FE	255	FF	1	01	1	01

3. Detecting a zero divisor is simply a matter of comparing the divisor to zero before the division operation. This comparison can be done by adding just two instructions to the program, immediately following LDA #00. They are:

```
        CMP  20
        BEQ  DONE
```

where DONE references the final instruction, JMP E1A1.

EXPERIMENT 8. SUBROUTINES AND THE STACK

1. The stack program is, simply:

```
LDX  #FF     Put stack address into X
TXS           and transfer it to Stack Pointer
LDA  #00     Initialize the registers
LDX  #01
LDY  #02
PHA          Push Accumulator
TXA          Push X Register
PHA
TYA          Push Y Register
PHA
BRK
```

Note that all pushes must be performed via the Accumulator.

2. The before and after "pictures" of the stack are:

BEFORE	Location	AFTER
20	$01FB	BD
58	$01FC	E1
E2	$01FD	02
BD	$01FE	01
E1	$01FF	00

The address in the Stack Pointer is now

Stack Pointer = $FC (or, $01FC)

Since the Stack Pointer will be incremented before PLA,

After PLA, Accumulator = $02

4. This subroutine adds 5 to location $20:

```
CLC
LDA  20
ADC  #05
STA  20
RTS
```

This program will call the subroutine three times:

```
LDA  #00
STA  20
JSR  0300
JSR  0300
JSR  0300
BRK
```

EXPERIMENT 8 (CONT'D)

5. After 5 has been added to location $20 three times,

   Location $20 = $0F

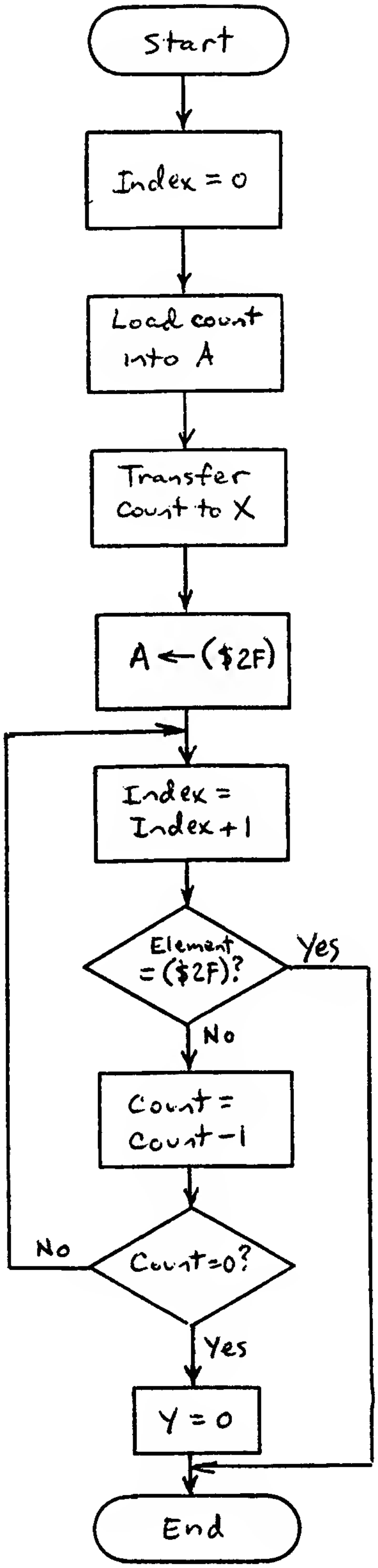
6. At power on, the AIM 65 Monitor initializes the Stack Pointer to this value:

   At power-on, Stack Pointer = $FF

   This is shown on lines 364 through 366 in the AIM 65 Monitor Program Listing.

EXPERIMENT 9. UNORDERED LISTS

1. The flowchart for the search program is shown below.


Start
Index = 0
Load count into A
Transfer Count to X
A ← ($2F)
Index = Index + 1
Element = ($2F)?
Yes
No
Count = Count − 1
Count = 0?
No
Yes
Y = 0
End

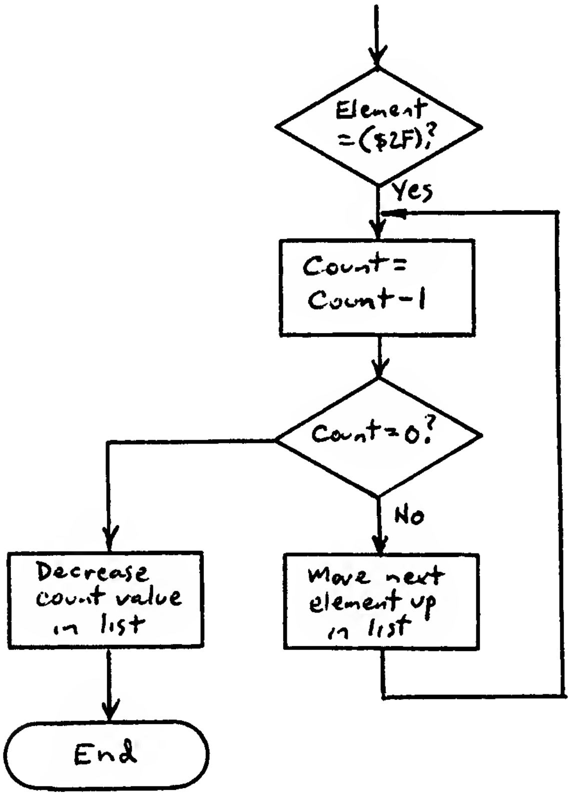
2. This program will return the element number in Y if the search value is found, and will return Y = 0 if the search value is not in the list:

```
        LDY   #00      Fetch element count
        LDA   (30),Y
        TAX             and put it in X
        LDA   2F       Load search value into Accumulator
NEXT    INY            Index to next element
        CMP   (30),Y   Element = ($2F)?
        BEQ   DONE     Yes.  Return with index in Y
        DEX            No.  Decrement element count
        BNE   NEXT     More elements to compare?
        LDY   #00      No.  Value was not found
DONE    JMP   E1A1     Return with indicator in Y
```

3. The correct initial values are:

Search Value ($2F) = $55
List Address, Low ($30) = $40
List Address, High ($31) = $00

4. The portion of the flowchart that deletes an element from the list looks like this:


Element =($2F)?
Yes
Count= Count-1
Count=0?
No
Decrease count value in list
Move next element up in list
End

EXPERIMENT 9 (CONT'D)

5. A program that will perform the "search-and-destroy" mission is:

```
        LDY  #00     Fetch element count
        LDA  (30),Y
        TAX           and put it in X
        LDA  2F      Load search value into Accumulator
NEXT    INY          Index to next element
        CMP  (30),Y  Element = ($2F)?
        BEQ  KILL    Yes. Delete this element
        DEX          No.  Decrement element count
        BNE  NEXT    More elements to compare?
        JMP  E1A1    No.  Return to Monitor
KILL    DEX          Decrement element count
        BEQ  DECCNT  End of list?
        INY          No.  Move next element up
        LDA  (30),Y
        DEY
        STA  (30),Y
        INY
        JMP  KILL
DECCNT  LDA  (30,X)  Decrease element count in list
        SBC  #01
        STA  (30,X)
        JMP  E1A1
```

6. The initial values that must be stored into memory are:

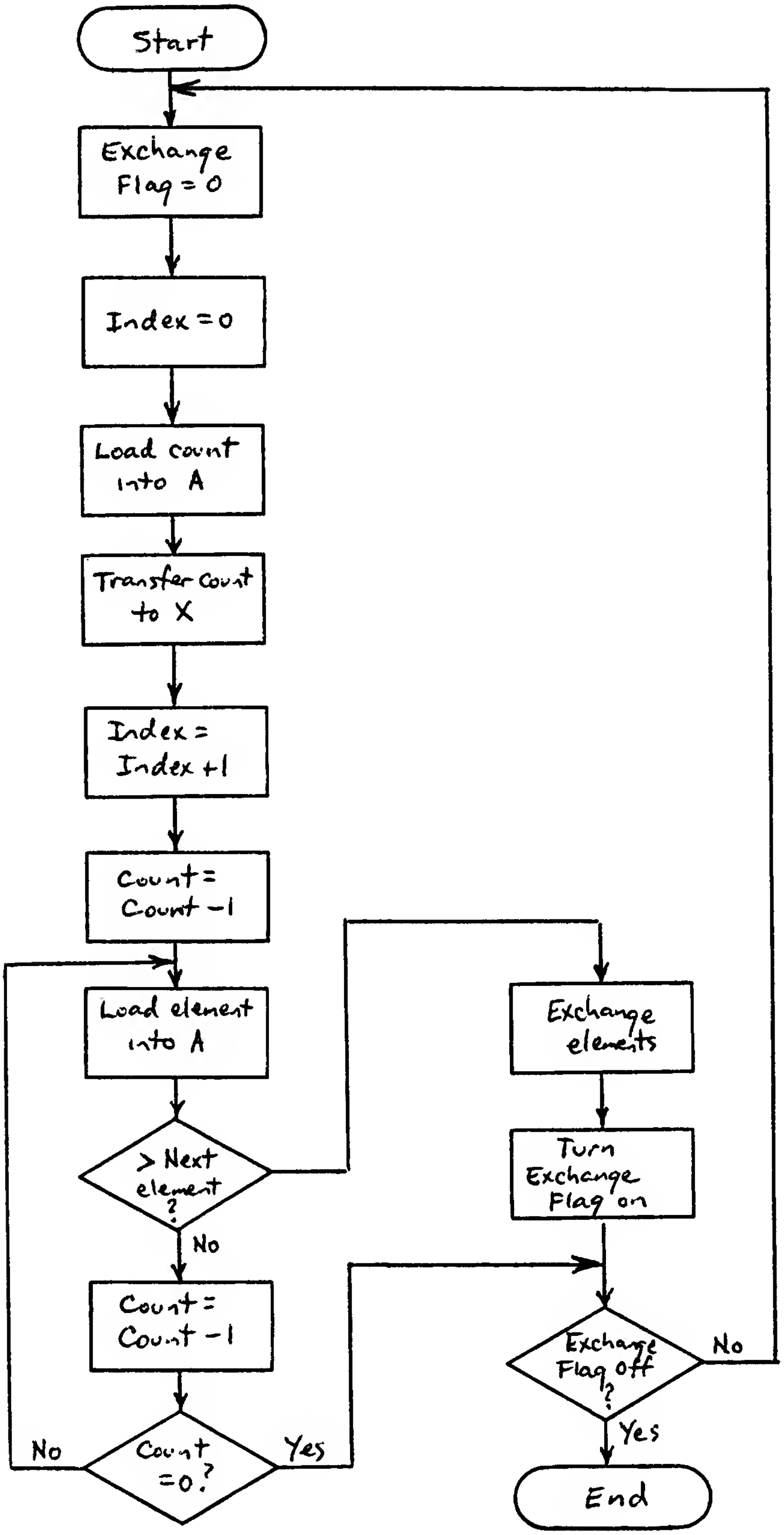
Search value ($2F) = $55
List address, low ($30) = $40
List address, high ($31) = $00

The search value is contained in location $48. After it has been deleted, the requested locations are:

($40) = $09 (Element count is now 9, rather than 10)
($47) = $FA
($48) = $06
($49) = $4F
($4A) = $4F

EXPERIMENT 10. SORTING UNORDERED DATA

1. The bubble sort flowchart will look something like this:


Start
Exchange Flag = 0
Index = 0
Load count into A
Transfer count to X
Index = Index +1
Count = Count −1
Load element into A
> Next element ?
No
Count = Count −1
Count = 0 ?
No
Yes
Exchange elements
Turn Exchange Flag on
Exchange Flag off ?
No
Yes
End

EXPERIMENT 10 (CONT'D)

Code for the bubble sort program is:

```
SORT8  LDY  #00     Turn off exchange flag (= 0)
       STY  32       and set index = 0
       LDA  (30),y  Fetch element count
       TAX           and put it into X
       INY          Point to first element
       DEX          Decrement count
NEXT   LDA  (30),Y  Fetch element
       INY
       CMP  (30),Y  Is it larger than next element?
       BCC  CHKEND
       BEQ  CHKEND
                    Yes.  Exchange elements
       PHA           by saving low byte on stack;
       LDA  (30),Y   then get high byte
       DEY           and store it at low address
       STA  (30),Y
       PLA          Pull low byte from stack
       INY           and store it at high address
       STA  (30),Y
       DEC  32      Turn on exchange flag (= -1)
CHKEND DEX          End of list?
       BNE  NEXT    No.  Fetch next element
       BIT  32      Yes.  Exchange flag still off?
       BMI  SORT8   No.  Go through list again
       JMP  E1A1    Yes.  List is now sorted
```

3. When the list has been sorted, it should be ordered like this:

LOCATION	CONTENTS
$40	$0A
$41	$00
$42	$06
$43	$40
$44	$40
$45	$4F
$46	$55
$47	$5A
$48	$A2
$49	$CE
$4A	$FA

EXPERIMENT 11. CODE CONVERSION FROM INPUT

1. A program that processes a single decimal digit from the keyboard is:

```
NEWDIG JSR  E973    Read digit
       CMP  #30     Character less than $30?
       BCC  NEWDIG
       CMP  #3A     No.  Is is greater than $39?
       BCS  NEWDIG
       AND  #0F     No.  Mask out the 4 MSB's
       JMP  E1A1     and return
```

2. The correct answers for the three runs are:

Press Key(s)	Accumulator
9	$09
N, then 4	$04
2	$02

(N is ignored)

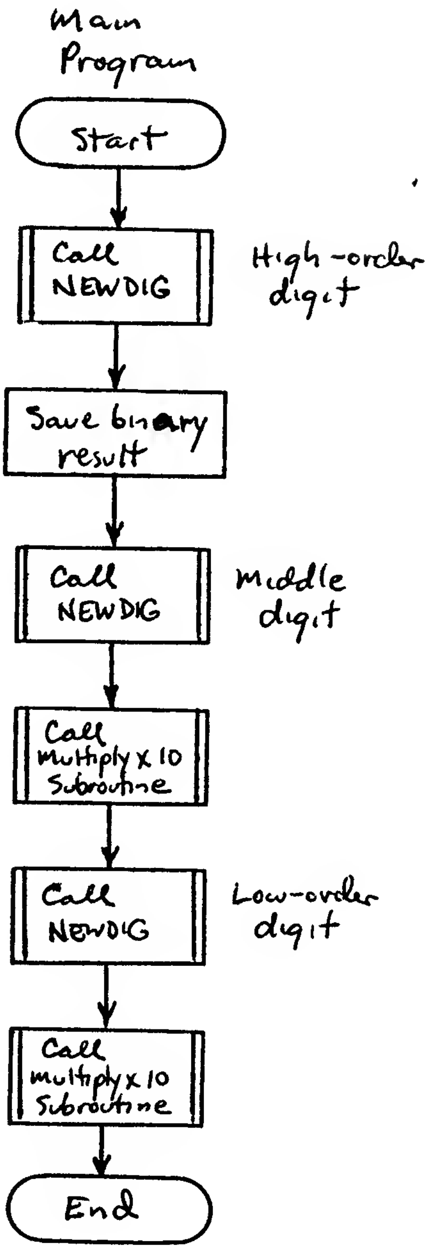
3. The flowchart is shown on the next page. The program code is:

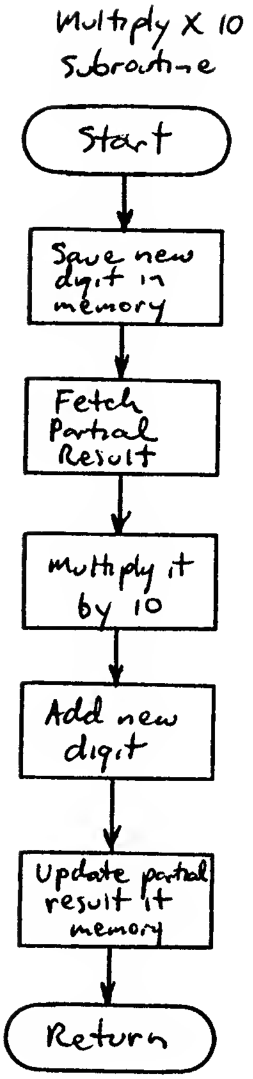
```
       JSR  NEWDIG  Fetch high-order digit
       STA  30       and store it as partial result
       JSR  NEWDIG  Fetch middle digit
       JSR  MULT10   and add it to partial result
       JSR  NEWDIG  Fetch low-order digit
       JSR  MULT10   and form final result
       BRK
        .
        .
        .
   NEWDIG (Step 1), in subroutine form goes here
        .
        .
        .
   The subroutine to follow multiplies the partial result
   by 10, then adds the new digit to the least significant
   digit position.

MULT10 STA  31      Save digit
       LDA  30      Fetch partial result
       ASL  A       Multiply it by two
       ASL  A       Multiply it by two again (Total = x 4)
       ADC  30      Add original result to it (Total = x 5)
       ASL  A       Multiply this by two (Total = x 10)
       ADC  31      Add new digit
       STA  30       and update partial result
       RTS
```

EXPERIMENT 11 (CONT'D)

Flowchart for Step 3 is:


Main Program
Start
Call NEWDIG
High-order digit
Save binary result
Call NEWDIG
Middle digit
Call multiply x 10 Subroutine
Call NEWDIG
Low-order digit
Call multiply x 10 Subroutine
End


Multiply X 10 Subroutine
Start
Save new digit in memory
Fetch Partial Result
multiply it by 10
Add new digit
Update partial result it memory
Return

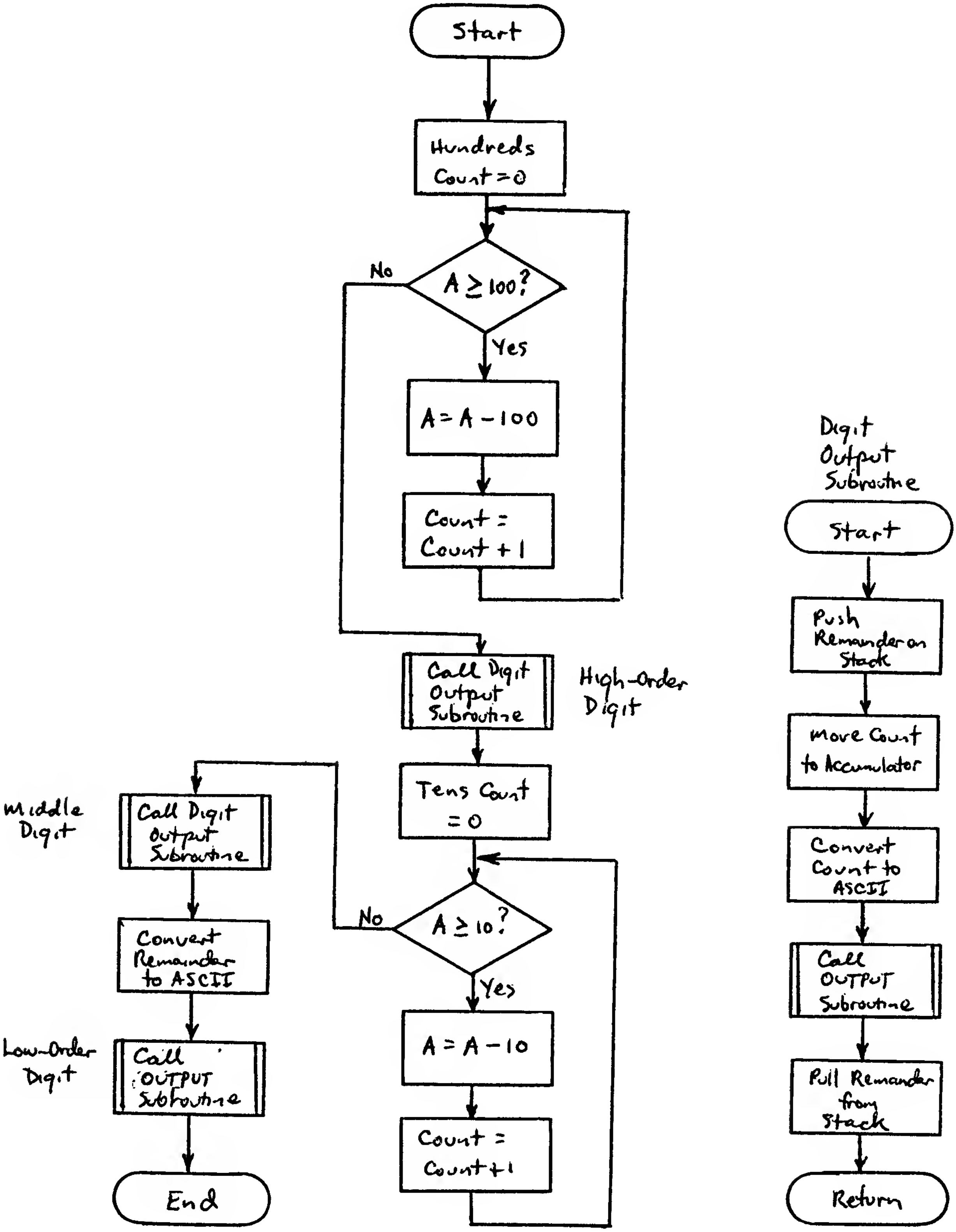
EXPERIMENT 11 (CONT'D)

4. The answers to the four test cases are:

Entered Number	Result (Hex.)	Carry? (Yes/No)
000	00	No
120	78	No
255	FF	No
256	00	Yes

EXPERIMENT 12. CODE CONVERSION FOR OUTPUT

1. The flowchart should look like this:


Start
Hundreds Count = 0
A ≥ 100?
No
Yes
A = A − 100
Count = Count + 1
Call Digit Output Subroutine
High-Order Digit
Tens Count = 0
A ≥ 10?
No
Yes
A = A − 10
Count = Count + 1
Middle Digit
Call Digit Output Subroutine
Convert Remainder to ASCII
Low-Order Digit
Call OUTPUT Subroutine
End
Digit Output Subroutine
Start
Push Remainder on Stack
Move Count to Accumulator
Convert Count to ASCII
Call OUTPUT Subroutine
Pull Remainder from Stack
Return

EXPERIMENT 12 (CONT'D)

This program will convert the binary value in the Accumulator to a three-digit ASCII decimal string:

```
        LDX  #00      Initialize hundreds counter
C100    CMP  #64      Binary value = or greater than 100?
        BCC  OUT1     No.  Go print decimal digit
        SBC  #64      Yes.  Subtract 100
        INX           Increment decimal count
        JMP  C100      and compare again
OUT1    JSR  PUTOUT   Go print hundreds count
        LDX  #00      Initialize tens counter
C10     CMP  #0A      Binary value = or greater than 10?
        BCC  OUT2     No.  Go print decimal digit
        SBC  #0A      Yes.  Subtract 10
        INX           Increment decimal counter
        JMP  C10       and compare again
OUT2    JSR  PUTOUT   Go print tens count
        CLC           Convert remainder to ASCII
        ADC  #30
        JSR  E97A      and print it with OUTPUT subroutine
        BRK
```

The print-out subroutine follows

```
PUTOUT  PHA           Save remainder on stack
        TXA           Move decimal count to Accumulator,
        ADC  #30       convert it to ASCII
        JSR  E97A      and print it
        PLA           Retrieve remainder from stack
        RTS
```

2. The results should be:

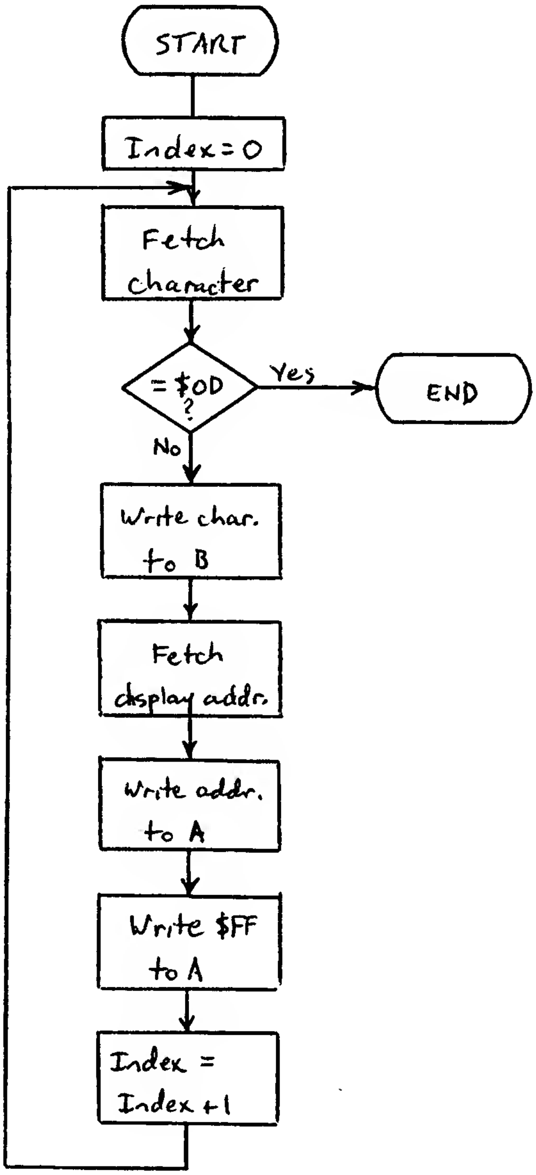
Number in Accumulator		Printed Result
Dec.	Hex.	
030	1E	030
255	FF	255
002	02	002
126	7E	126

EXPERIMENT 13. INPUT/OUTPUT

1. Program to display the letter M.

```
        LDA #C1    Load character code for M
        STA AC02    and write it to B
        LDA #7B    Load display address 1
        STA AC00    and write it to A
        LDA #FF    Deselect display
        STA AC00
  HERE  JMP HERE   Wait for Reset
```

2. Flowchart for name display program


START
Index=0
Fetch character
=$0D ?
Yes
END
No
Write char. to B
Fetch display addr.
Write addr. to A
Write $FF to A
Index = Index+1

EXPERIMENT 13 (CONT'D)

Here is the code for the name display program:

```
      LDX  #00     Index = 0
LOOP  LDY  0300,X  Fetch character
      CPY  #0D     End of name table?
WAIT  BEQ  WAIT    Yes.  Wait for reset
      STY  AC02    No.  Write character to B
      LDA  0400,X  Fetch display address
      STA  AC00     and write it to A
      LDA  #FF     Deselect display
      STA  AC00
      INX          Increment index
      JMP  LOOP     and return for next character
```

EXPERIMENT 14. A MORE POWERFUL I/O DEVICE, THE R6522 VIA

The answers to questions in the Discussion are:

A. Auxiliary Control Register ($A00B) = $40
This value is used to select the Timer 1 free-running mode, without output on PB7

B. For a 10-millisecond time interval
Timer 1 Latch, low = $0E
Timer 1 Latch, high = $27

1. The time program should look like this:

```
      LDA  #00    Timeout Count = 0
      STA  20
      STA  21
      LDA  #40    Select free-running mode
      STA  A00B
      LDA  #0E    Load count into Latches
      STA  A004
      LDA  #27
      STA  A005
LOOP  BIT  A00D   Timeout?
      BVC  LOOP   No.  Wait for timeout
      INC  20     Yes.  Increment count
      BNE  CLEAR
      INC  21
CLEAR LDA  A004   Clear Timer 1 Interrupt Flag
      JMP  LOOP
```

Students should be encouraged to use a BIT instruction at LOOP (as done here), rather than some logical operation, because BIT will reflect the status of Bit 6 in Overflow.

3. Maximum total time (seconds) = 655.35 seconds
This answer is obtained by realizing that two memory locations can hold values of up to 65,535 (or $FFFF), and the count is expressed here in hundredths of a second.

EXPERIMENT 15. INTERRUPTS

The answer to the question in the Discussion is:

Interrupt Enable Register ($A00E) = $C0

since Bit 7 = 1 to enable the lower-order bits and Bit 6 = 1 to specifically enable the Timer 1 interrupt.

1. The main program should look like this:

```
           LDA   #00       Interrupt Vector = $0300
           STA   A400
           LDA   #03
           STA   A401
           LDA   #C0       Enable Timer 1 interrupt
           STA   A00E
           CLI             Enable CPU interrupts
           LDA   #00       Timeout Count = 0
           STA   20
           STA   21
           LDA   #40       Select free-running mode
           STA   A00B
           LDA   #0E       Load count into Latches
           STA   A004
           LDA   #27
           STA   A005
     HERE  JMP   HERE
```

2. The interrupt service routine should look like this:

```
           INC   20        Increment count
           BNE   CLEAR
           INC   21
     CLEAR LDA   A004      Clear Timer 1 Interrupt Flag
           RTI             Return from interrupt
```

4. After pressing Reset

   Auxiliary Control Register ($A00B) = $00

   This value selects the one-shot mode, with no output on PB7.

EXPERIMENT 16. A TIMING PROGRAM WITH DECIMAL OUTPUT

1. To maintain the count in BCD form, the interrupt service routine should look like this:

```
      SED             Set Decimal Mode
      CLC             Add 1 to count
      LDA   20
      ADC   #01
      STA   20
      LDA   21
      ADC   #00
      STA   21
      CLD             Clear Decimal Mode
      LDA   A004      Clear Timer 1 Interrupt Flag
      RTI
```

2. Because the time count consists of four BCD digits,

   Maximum total time (seconds) = 99.99

3. With display, the interrupt service routine should take this form:

```
      SED
      CLC
      LDA   20
      ADC   #01
      STA   20
      LDA   21
      ADC   #00
      STA   21
      CLD
      LDA   A004
      JSR   EB44      Call CLR, to clear display
      LDA   21        Fetch seconds count
      LSR   A         Shift it into the 4 LSB's
      LSR   A
      LSR   A
      LSR   A
      JSR   CONVT     Convert and output high digit
      LDA   21        Fetch seconds count again
      AND   #0F       Mask off 4 MSB's
      JSR   CONVT     Convert and output low digit
      RTI             Return from interrupt
CONVT CLC             Convert BCD digit to ASCII
      ADC   #30
      JMP   E97A       and output it, with OUTPUT
```

EXPERIMENT 17. THE AIM 65 ASSEMBLER

2. The proper responses are:

```
EDITOR
FROM=0200   TO=040E
IN=(SPACE)
```

3. Here is a listing of the source code, in the form it is entered into the Text Buffer. Note that you can always begin entering information in the leftmost position on the line, and that labels, mnemonics and operands must be separated by at least one space.

```
*=$500
LDA $21
CMP $20
BCC SWAP
BEQ EQUAL
LDA #0
JMP STORE
EQUAL LDA #1
JMP STORE
SWAP LDX $20
STX $21
STA $20
LDA #2
STORE STA $22
JMP $E1A1
 END
```

4. The responses to the N (Assembler) prompts are:

```
ASSEMBLER
FROM=800    TO=1000
IN=M
LIST?Y
LIST-OUT=P

OBJ?N
```

The LIST-OUT prompt could have been answered by pressing RETURN or SPACE instead of P. The result would have been the same.

Further, if you wanted just an errors-only list, rather than the full assembly listing, you could have answered "N" in response to the LIST? prompt.

EXPERIMENT 17 (CONT'D)

Here is the assembly listing that will be produced. Note that the hexadecimal codes for each instruction are produced in a column at the left of the listing.

```
==0000
       *=$500
==0500
A521   LDA $21
C520   CMP $20
900C   BCC SWAP
F005   BEQ EQUAL
A900   LDA #0
4C1A05 JMP STORE
==050D EQUAL
A901   LDA #1
4C1A05 JMP STORE
==0512 SWAP
A620   LDX $20
8621   STX $21
8520   STA $20
A902   LDA #2
==051A STORE
8522   STA $22
4CA1E1 JMP $E1A1
       .END
 ERRORS= 0000
```

5. The correct answer is:

   To re-enter the Text Buffer, press T.

   Don't press E, or you'll lose the contents of the Text Buffer!

7. As in Experiment 4:

   ($20) = $0E   ($21) = $9C   ($22) = $02